普通高等教育"十一五"国家级规划教材

清華大學 计算机系列教材

张尧学 编著

计算机操作系统教程（第4版）

习题解答与实验指导



清华大学出版社

清华大学计算机系列教材
普通高等教育"十一五"国家级规划教材

计算机操作系统教程

（第 4 版）

习题解答与实验指导

张尧学　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本书是作者在清华大学计算机系多年教学经验和科研成果的基础上，配合清华大学计算机系列教材之一的《计算机操作系统教程》(第 4 版)而编写的相关习题解答和实验指导。全书分为两大部分：第一部分是《计算机操作系统教程》(第 4 版)中各章习题的参考解答和部分硕士研究生入学考试用题及解答；第二部分为清华大学计算机系操作系统课程教学用实验指导及相应的程序设计与源代码分析。实验主要设计在 Linux 环境下用 C 语言编程完成，也可在 UNIX 系统Ⅴ或其他更高版本的 UNIX 环境下完成。

本书既可作为计算机专业和其他相关专业操作系统课程的补充教材，也可供有关人员自学，或供操作系统等系统设计人员阅读和参考。

图书在版编目(CIP)数据

计算机操作系统教程习题解答与实验指导/张尧学编著. --4 版. --北京：清华大学出版社，2013
清华大学计算机系列教材
ISBN 978-7-302-33675-4

Ⅰ.①计…　Ⅱ.①张…　Ⅲ.①操作系统—高等学校—教学参考资料　Ⅳ.①TP316

中国版本图书馆 CIP 数据核字(2013)第 206346 号

责任编辑：白立军　战晓雷
封面设计：常雪影
责任校对：时翠兰
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京国马印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：9.5　　**字　　数**：227 千字
版　　次：1993 年 11 月第 1 版　2013 年 11 月第 4 版　　**印　　次**：2013 年 11 月第1 次印刷
印　　数：1～3000
定　　价：22.00 元

产品编号：055443-01

序

“清华大学计算机系列教材”已经出版发行了30余种，包括计算机科学与技术专业的基础数学、专业技术基础和专业等课程的教材，覆盖了计算机科学与技术专业本科生和研究生的主要教学内容。这是一批至今发行数量很大并赢得广大读者赞誉的书籍，是近年来出版的大学计算机专业教材中影响比较大的一批精品。

本系列教材的作者都是我熟悉的教授与同事，他们长期在第一线担任相关课程的教学工作，是一批很受本科生和研究生欢迎的任课教师。编写高质量的计算机专业本科生（和研究生）教材，不仅需要作者具备丰富的教学经验和科研实践，还需要对相关领域科技发展前沿的正确把握和了解。正因为本系列教材的作者们具备了这些条件，才有了这批高质量优秀教材的产生。可以说，教材是他们长期辛勤工作的结晶。本系列教材出版发行以来，从其发行的数量、读者的反映、已经获得的国家级与省部级的奖励，以及在各个高等院校教学中所发挥的作用上，都可以看出本系列教材所产生的社会影响与效益。

计算机学科发展异常迅速，内容更新很快。作为教材，一方面要反映本领域基础性、普遍性的知识，保持内容的相对稳定性；另一方面，又需要紧跟科技的发展，及时地调整和更新内容。本系列教材都能按照自身的需要及时地做到这一点。如王爱英教授等编著的《计算机组成与结构》、戴梅萼教授等编著的《微型计算机技术及应用》都已经出版了第四版，严蔚敏教授的《数据结构》也出版了三版，使教材既保持了稳定性，又达到了先进性的要求。

本系列教材内容丰富，体系结构严谨，概念清晰，易学易懂，符合学生的认知规律，适合教学与自学，深受广大读者的欢迎。系列教材中多数配有丰富的习题集、习题解答、上机及实验指导和电子教案，便于学生理论联系实际地学习相关课程。

随着我国进一步的开放，我们需要扩大国际交流，加强学习国外的先进经验。在大学教材建设上，我们也应该注意学习和引进国外的先进教材。但是，“清华大学计算机系列教材”的出版发行实践以及它所取得的效果告诉我们，在当前形势下，编写符合国情的具有自主版权的高质量教材仍具有重大意义和价值。它与国外原版教材不仅不矛盾，而且是相辅相成的。本系列教材的出版还表明，针对某一学科培养的要求，在教育部等上级部门的指导下，有计划地组织任课教师编写系列教材，还能促进对该学科科学、合理的教学体系和内容的研究。

我希望今后有更多、更好的我国优秀教材出版。

清华大学计算机系教授，中国科学院院士

张钹

第 4 版前言

计算机技术的飞速发展正在引发新的一轮世界性技术革命。在经济发展越来越全球化、科技创新越来越国际化、知识经济已初见端倪的今天,任何一门技术或任何一个领域离开了计算机都是不可想象的。而计算机技术发展之迅速,计算机及其相关 IT 产品市场竞争之激烈,计算机产业让人致富速度之迅猛,也同样是人们始料不及的。在 21 世纪,任何想在技术领域有一番作为的人,都不得不面对计算机技术的挑战。

软件技术是计算机系统的灵魂与核心,而操作系统更是计算机系统的大脑。“想发财,学软件!”在一些国家已成为深入人心的广告词。在我国,科技创新、高科技产业化的浪潮也势必会以雷霆万钧之力推动软件技术的迅猛发展与普及。21 世纪的哪一行哪一业能够离开软件呢?

学习计算机软件技术,特别是计算机操作系统技术,除了需要刻苦努力外,还需要掌握软件和操作系统的原理与设计技巧。这些原理与技巧可以说是计算机界的前辈们一代接一代不停顿的努力所留下的知识与智慧的结晶,学习和掌握它们对于激发自己的创造力和想象力是很有帮助的。

如何学习和掌握操作系统技术的原理与实际技巧呢?除了听课和读书之外,最好的方法就是在实践中练习。例如,自己设计一个小型操作系统,多使用操作系统,多阅读和分析操作系统源代码等。当前非常流行的 Linux 操作系统的原始版事实上也是一位优秀的大学生的练习之作。除了上述练习方法之外,习题和实验也是很重要的实践之一。

本书是配合《计算机操作系统教程》(第 4 版)的习题解答与实验指导书。本书除给出《计算机操作系统教程》(第 4 版)各章所附习题的参考答案外,还给出一些有关的综合试题及其参考答案;另外,还设计了 4 个在 Linux 环境下或 UNIX 环境下的小实验,包括进程控制、进程通信、内存管理以及文件系统设计等,并给出了这 4 个实验的参考编程解答。

本书的编写得到了清华大学计算机系网络系统组杨华杰的大力支持和帮助,她对本书中的部分习题进行了解答和完善,而且重新编写了实验程序。

本书虽然给出了《计算机操作系统教程》(第 4 版)一书中习题的参考解答和相关实验指导,但由于作者的水平与知识所限,这些解答只是一种参考,里面完全可能存在错误和不妥之处,有待于有识之士的指教。此外,还希望读者不要局限于这些解答。

衷心希望本书能对学习计算机操作系统和计算机软件的人们有所帮助!

作者

2013 年 6 月于清华园

目　　录

第一部分

习 题 解 答

第1章　绪　　论

1.1　什么是操作系统的基本功能？

答：操作系统的职能是管理和控制计算机系统中的所有硬件和软件资源，合理地组织计算机工作流程，并为用户提供一个良好的工作环境和友好的接口。操作系统的基本功能包括处理机管理、存储管理、设备管理、信息管理（文件系统管理）和用户接口等。

1.2　什么是批处理、分时和实时系统？它们各有什么特征？

答：批处理系统（batch processing operating system）：操作员把用户提交的作业分类，把一批作业编成一个作业执行序列，由专门编制的监督程序（monitor）自动依次处理。其主要特征是用户脱机使用计算机、成批处理以及多道程序运行。

分时系统（time sharing operating system）：把处理机的运行时间分成很短的时间片，按时间片轮转的方式把处理机分配给各进程使用。其主要特征是交互性、多用户同时性和独立性。

实时系统（real time operating system）：在被控对象允许时间范围内作出响应。其主要特征是：对实时信息分析处理速度要比进入系统快，要求安全可靠，资源利用率低。

1.3　多道程序（multiprogramming）和多重处理（multiprocessing）有何区别？

答：多道程序是作业之间自动调度执行、共享系统资源，并不是真正地同时执行多个作业；而多重处理系统配置多个CPU，能真正同时执行多道程序。要有效使用多重处理，必须采用多道程序设计技术，而多道程序设计原则上不一定要求多重处理系统的支持。

1.4　讨论操作系统可以从哪些角度出发？如何把它们统一起来？

答：讨论操作系统可以从以下角度出发：(1)操作系统是计算机资源的管理者；(2)操作系统为用户提供使用计算机的界面；(3)用进程管理观点研究操作系统，即围绕进程运行过程来讨论操作系统。

上述这些观点彼此并不矛盾，分别代表了从不同角度看待同一事物（操作系统）的观点。每一种观点都有助于理解、分析和设计操作系统。

1.5　写出1.6节中巡回置换算法的执行结果。

答：1.6节中的巡回置换算法要求：

设 i=1,2,3,4,5,6,7

　　p[i]= 4,7,3,1,2,5,6

当 k∈[1…n]

　　k=P[…, p[k], …]。

从而有如下解。

（1）算法如下：

```
local x, k             /* x和k为局部变量 */
begin
     k←1               /* 初始化k */
     while k<=7 do
          x←k
          repeat
               print(x)
               x←p[x]
          until x=k
          k←k+1
     od
end
```

（2）打印结果如下。

k＝1时，置换过程为：1 4 1。

k＝2时，置换过程为：2 7 6 5 2。

k＝3时，置换过程为：3 3。

k＝4时，置换过程为：4 1 4。

k＝5时，置换过程为：5 2 7 6 5。

k＝6时，置换过程为：6 5 2 7 6。

k＝7时，置换过程为：7 6 5 2 7。

1.6　设计计算机操作系统与哪些硬件器件有关？

答：计算机操作系统的重要功能之一是对硬件资源的管理。因此设计计算机操作系统时应考虑下述计算机硬件资源：

（1）CPU与指令的长度及执行方式；

（2）内存、缓存和高速缓存等存储装置；

（3）各类寄存器，包括各种通用寄存器、控制寄存器和状态寄存器等；

（4）中断机构；

（5）外部设备与I/O控制装置；

（6）内部总线与外部总线；

（7）对硬件进行操作的指令集。

第 2 章　操作系统用户界面

2.1　什么是作业和作业步?

答:把在一次应用业务处理过程中,从输入开始到输出结束,用户要求计算机所做的有关该次业务处理的全部工作称为一个作业。从系统的角度看,作业则是一个比程序更广的概念。作业由程序、数据和作业说明书组成。系统通过作业说明书控制文件形式的程序和数据,使之执行和操作。而且,在批处理系统中,作业是抢占内存的基本单位。也就是说,批处理系统以作业为单位把程序和数据调入内存以便执行。作业由顺序相连的不同作业步组成。

作业步是在一个作业的处理过程中计算机所做的相对独立的工作。例如,编辑输入是一个作业步,它产生源程序文件;编译也是一个作业步,它产生目标代码文件。

2.2　作业由哪几部分组成?这几部分各有什么功能?

答:作业由 3 部分组成:程序、数据和作业说明书。程序和数据完成用户所要求的业务处理工作,系统通过作业说明书控制文件形式的程序和数据,使之执行和操作。

2.3　作业的输入方式有哪几种?各有何特点?

答:作业的输入方式有 5 种:联机输入方式、脱机输入方式、直接耦合方式、Spooling (Simultaneous Peripheral Operations Online)系统和网络联机方式,这 5 种输入方式各有如下特点。

(1) 联机输入方式:用户和系统通过交互式会话来输入作业。

(2) 脱机输入方式:利用低档个人计算机作为外围处理机进行输入处理,存储在后援存储器上,然后将此后援存储器连接到高速外围设备上和主机相连,从而在较短的时间内完成作业的输入工作。

(3) 直接耦合方式:把主机和外围低档机通过一个公用的大容量外存直接耦合起来,从而省去了在脱机输入中那种依靠人工干预来传递后援存储器的过程。

(4) Spooling 系统:可译为外围设备同时联机操作。在 Spooling 系统中,多台外围设备通过通道或 DMA 器件和主机与外存连接起来,作业的输入输出过程由主机中的操作系统控制。

(5) 网络联机方式:以上述几种输入输出方式为基础。当用户通过计算机网络中的某一台设备对计算机网络中的另一台主机进行输入输出操作时,就构成了网络联机方式。

2.4　试述 Spooling 系统的工作原理。

答:在 Spooling 系统中,多台外围设备通过通道或 DMA 器件和主机与外存连接起来,作业的输入输出过程由主机中的操作系统控制。操作系统中的输入程序包含两个独立的过程,一个过程负责从外部设备把信息读入缓冲区;另一个过程是写过程,负责把缓冲区中的

信息送入到外存输入井中。

在系统输入模块收到作业输入请求后，输入管理模块中的读过程负责将信息从输入装置读入缓冲区。当缓冲区满时，由写过程将信息从缓冲区写到外存输入井中。读过程和写过程反复循环，直到一个作业输入完毕。当读过程读到一个硬件结束标志后，系统再次驱动写过程把最后一批信息写入外存并调用中断处理程序结束该次输入。然后，系统为该作业建立作业控制块(JCB)，从而使输入井中的作业进入作业等待队列，等待作业调度程序选中后进入内存。

2.5　操作系统为用户提供哪些接口？它们的区别是什么？

答：操作系统为用户提供两个接口。一个是系统为用户提供的各种命令接口，用户利用这些操作命令来组织和控制作业的执行或管理计算机系统。另一个接口是系统调用，编程人员使用系统调用来请求操作系统提供服务，例如申请和释放外设等类资源、控制程序的执行速度等。

2.6　作业控制方式有哪几种？调查你周围的计算机的作业控制方式。

答：作业控制的主要方式有两种：脱机方式和联机方式。

脱机控制方式利用作业控制语言来编写表示用户控制意图的作业控制程序，也就是作业说明书。作业控制语言的语句就是作业控制命令。不同的批处理系统提供不同的作业控制语言。

联机控制方式不同于脱机控制方式，它不要求用户填写作业说明书，系统只为用户提供一组键盘或其他操作方式的命令。用户使用操作系统提供的操作命令和系统会话，交互地控制程序执行和管理计算机系统。

2.7　什么是系统调用？系统调用与一般用户程序有什么区别？与库函数和实用程序又有什么区别？

答：系统调用是操作系统提供给编程人员的唯一接口。编程人员利用系统调用，在源程序一级动态请求和释放系统资源，调用系统中已有的系统功能来完成那些与机器硬件部分相关的工作以及控制程序的执行速度等。因此，系统调用像一个黑箱子那样，对用户屏蔽了操作系统的具体动作而只提供有关的功能。它与一般用户程序、库函数和实用程序的区别是：系统调用程序是在核心态执行，调用它们需要一个类似于硬件中断处理的中断处理机制来提供系统服务。

2.8　简述系统调用的实现过程。

答：用户在程序中使用系统调用，给出系统调用名和函数后，即产生一条相应的陷入指令，通过陷入处理机制调用服务，引起处理机中断，然后保护处理机现场，取系统调用功能号并寻找子程序入口，通过入口地址表来调用系统子程序，然后返回用户程序继续执行。

2.9　为什么说分时系统没有作业的概念？

答：因为在分时系统中，每个用户得到的时间片有限，用户的程序和数据信息直接输入

到内存工作区中和其他程序一起抢占系统资源投入执行，而不必进入外存输入井等待作业调度程序选择。因此，分时系统没有作业控制表，也没有作业调度程序。

2.10　Linux 操作系统为用户提供哪些接口？试举例说明。

答：Linux 系统为用户提供两种接口，一种是面向操作命令的接口 Shell，另一种是面向编程用户的接口，即系统调用。常见的 Shell 命令有 login，logout，vi，emacs，cp，rm，ls，cc，link，adduser，chown，dbx，date 等；常见的系统调用有 ioctl，read，write，open，close，creat，execl，flock，stat，mount，fork，wait，exit，socket 等。

2.11　在你周围装有 Linux 系统的计算机上，查看有关 Shell 的基本命令，并编写一个简单的 Shell 程序，完成一个已有数据文件的复制和打印。

答：假设需要将文件 src.txt 复制为 dst.txt，Shell 程序如下：

```
#!/bin/bash
#copy file
cat src.txt> dst.txt
#print file
cat src.txt> /dev/lp
```

2.12　用 Linux 文件读写的相关系统调用编写 copy 程序。

答：假设 copy 程序的执行格式为 copy src dst，程序的代码如下：

```
#include < sys/types.h>
#include < sys/stat.h>
#include < fcntl.h>
#include < unistd.h>
#define BUFSIZE 8192
int main(char ** argv, int argc)
{
    if (argc!= 3)
    {
        print("\n usage: copy src dst\n");
        return - 1;
    }
    int src,dst;
    char buf[BUFSIZE];
    int n;

    src= open(argv[1], O_RDONLY);
    dst= open(argv[2], O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IXUSR);
    while ((n= read(src,buf,BUFSIZE))> 0)
    {
        if (write(dst,buf,n)!= n)
            print("write error!")
```

```
    }
    if (n< 0)
        print("read error!");
    close(src);
    close(dst);
    exit(0);
}
```

2.13 用 Windows 的 dll 接口编写 copy 程序。

答：（1）实现 dll 的程序：

```
dlltest.cpp
#include "windows.h"
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}

extern "C" _declspec(dllexport) int MyCopyFile(LPCSTR src,LPCSTR tar)
{
    if (CopyFile(src,tar,FALSE)= = TRUE)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

（2）测试 dll 的程序：

```
#include "windows.h"

extern "C" _declspec (dllimport) int MyCopyFile(LPCSTR,LPCSTR);
int main(int argc, char * argv[])
{
    MyCopyFile("C:\\1.txt","C:\\2.txt");
    return 0;
}
```

第 3 章　进 程 管 理

3.1　有人说,一个进程是由伪处理机执行的一个程序,这话对吗？为什么？

答：对。

因为伪处理机的概念只有在执行时才存在,它表示多个进程在单处理机上并发执行的一个调度单位。因此,尽管进程是动态概念,是程序的执行过程,但是,在多个进程并行执行时,仍然只有一个进程占据处理机执行,而其他并发进程则处于就绪或等待状态。这些并发进程就相当于由伪处理机执行的程序。

3.2　试比较进程和程序的区别。

答：(1) 进程是一个动态概念,而程序是一个静态概念,程序是指令的有序集合,无执行含义,进程则强调执行的过程。

(2) 进程具有并行特征(独立性、异步性),程序则没有。

(3) 不同的进程可以包含同一个程序,同一程序在执行中也可以产生多个进程。

3.3　我们说程序的并发执行将导致最终结果失去封闭性。这话对所有的程序都成立吗？试举例说明。

答：并非对所有的程序均成立。例如：

```
begin
    local x
    x:= 10
    print(x)
end
```

上述程序中 x 是内部变量,不可能被外部程序访问,因此这段程序的运行不会受外部环境影响。

3.4　试比较作业和进程的区别。

答：一个进程是一个程序对某个数据集的执行过程,是分配资源的基本单位。作业是用户为了让计算机完成某项任务而要求计算机所做工作的集合。一个作业的完成要经过作业提交、作业收容、作业执行和作业完成 4 个阶段。而进程是已提交完毕的程序的执行过程的描述,是资源分配的基本单位。二者的主要区别如下：

(1) 作业是用户向计算机提交任务的任务实体。在用户向计算机提交作业之后,系统将它放入外存中的作业等待队列中等待执行。而进程则是完成用户任务的执行实体,是向系统申请分配资源的基本单位。任一进程,只要它被创建,总有相应的部分存在于内存中。

(2) 一个作业可由多个进程组成,且必须至少由一个进程组成,但反过来不成立。

(3) 作业的概念主要用在批处理系统中,像 UNIX 这样的分时系统中则没有作业的概

念。而进程的概念则用在几乎所有的多道程序系统中。

3.5　在 UNIX System Ⅴ 中，系统程序所对应的正文段未被考虑成进程上下文的一部分，为什么？

答：因为系统程序的代码被用户程序所共享，因此如果每个进程在保存进程上下文时，都将系统程序代码放到其进程上下文中，则大大浪费了资源。因此系统程序的代码不放在进程上下文中，而是统一放在核心程序所处的内存中。

3.6　什么是临界区？试举一个临界区的例子。

答：临界区是指不允许多个并发进程交叉执行的一段程序。它是由于不同并发进程的程序段共享公用数据或公用数据变量而引起的，所以它又被称为访问公用数据的那段程序。例如：

```
getspace:
begin
     local g
     g= stack[top]
     top= top- 1
end
release(ad):
begin
     top= top+ 1
     stack[top]= ad
end
```

3.7　并发进程间的制约有哪两种？引起制约的原因是什么？

答：并发进程所受的制约有两种：直接制约和间接制约。

直接制约是由并发进程互相共享对方的私有资源所引起的。间接制约是由竞争共有资源而引起的。

3.8　什么是进程间的互斥？什么是进程间的同步？

答：进程间的互斥是指：一组并发进程中的一个或多个程序段，因共享某一公有资源而导致它们必须以一个不许交叉执行的单位执行，即不允许两个以上的共享该资源的并发进程同时进入临界区。

进程间的同步是指：异步环境下的一组并发进程因直接制约互相发送消息而进行互相合作、互相等待，是各进程按一定的速度执行的过程。

3.9　试比较 P、V 原语法和加锁法实现进程间互斥的区别。

答：互斥的加锁实现是这样的：当某个进程进入临界区之后，它将锁上临界区，直到它退出临界区时为止。并发进程在申请进入临界区时，首先测试该临界区是否是上锁的，如果该临界区已被锁住，则该进程要等到该临界区开锁之后才有可能获得临界区。

但是加锁法存在如下弊端：(1)循环测试锁定位将损耗较多的 CPU 计算时间；(2)产生不公平现象。

为此，P、V 原语法采用信号量管理相应临界区的公有资源，信号量的数值仅能由 P、V 原语操作改变，而 P、V 原语执行期间不允许中断发生。其过程是这样的：当某个进程正在临界区内执行时，其他进程如果执行了 P 原语，则该进程并不像 lock 时那样因进不了临界区而返回到 lock 的起点，等以后重新执行测试，而是在等待队列中等待由其他进程做 V 原语操作释放资源后，进入临界区，这时 P 原语才算真正结束。若有多个进程做 P 原语操作而进入等待状态之后，一旦有 V 原语释放资源，则等待进程中的一个进入临界区，其余的继续等待。

总之，加锁法是采用反复测试 lock 而实现互斥的，存在 CPU 浪费和不公平现象，P、V 原语使用了信号量，克服了加锁法的弊端。

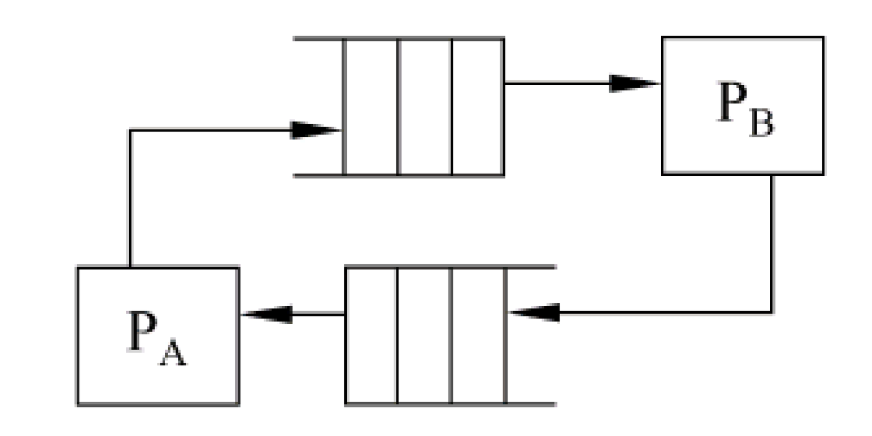
3.10　设在 3.6 节中所描述的生产者-消费者问题中，其缓冲部分由 *m* 个长度相等的有界缓冲区组成，且每次传输数据长度等于有界缓冲区长度，生产者和消费者可对缓冲区同时操作。重新描述发送过程 deposit(data)和接收过程 remove(data)。

答：设第 i 块缓冲区的公用信号量为 mutex[i]，保证生产者进程和消费者进程对同一块缓冲区操作的互斥，初始值为 1。设信号量 avail 为生产者进程的私用信号量，初始值为 m；信号量 full 为消费者进程的私用信号量，初始值为 0。从而有

```
deposit(data)
    begin
        P(avail)
        选择一个空缓冲区 i
        P(mutex[i])
        送数据入缓冲区 i
        V(full)
        V(mutex[i])
    end
remove(data)
    begin
        P(full)
        选择一个满缓冲区 i
        P(mutex[i])
        取缓冲区 i中的数据
        V(avail)
        V(mutex[i])
    end
```

3.11　两个进程 P_A 和 P_B 通过两个 FIFO 缓冲区队列连接(如下图所示)，每个缓冲区长度等于传送消息长度。进程 P_A 和 P_B 之间的通信满足如下条件：

(a) 至少有一个空缓冲区存在时，相应的发送进程才能发送一个消息。



(b) 当缓冲队列中至少存在一个非空缓冲区时，相应的接收进程才能接收一个消息。

试描述发送过程 send(i,m)和接收过程 receive(i,m)。这里 i 代表缓冲队列。

答：定义数组 buf[0]、buf[1]、bufempty[0]和 buffull[1]是 P_A 的私有信息量，buffull[0]、bufempty[1]是 P_B 的私有信息量。

初始时：

```
bufempty[0]= bufempty[1]= n,(n 为缓冲区队列的缓冲区个数)
buffull[0]= buffull[1]= 0
send(i,m)
begin
     local x
    P(bufempty[i])
    按 FIFO 方式选择一个空缓冲区
    buf[i](x)
    buf[i](x)= m
    buf[i](x)置满标记
    V(buffull[i])
end
receive(i,m)
begin
     local x
    P(buffull[i])
    按 FIFO 方式选择一个装满数据的缓冲区 buf[i](x)
    m= buf[i](x)
    buf[i](x)置空标记
    V(bufempty[i])
end
```

P_A 调用 send(0,m)和 receive(1,m)。

P_B 调用 send(1,m)和 receive(0,m)。

3.12 在和控制台通信的例中，设操作员不仅回答用户进程所提出的问题，而且还能独立地向各用户进程发出指示。对于这些指示，操作员不要求用户进程回答，但它们享有比其他消息优先传送的优先度。即如果 inbuf 中有指示存在，系统不能进行下一次通信会话。试按上述要求重新描述 CCP、KCP 和 DCP。

答：KCP 描述如下：

设 T_Ready 和 T_Busy 分别为键盘 KP 和键盘控制进程 KCP 的私有信号量，其初值为 0 和 1。设 inbuffer 为 inbuf 的共有信号量，初值为 1，表示其中没有控制消息。

```
初始化{清除所有 inbuf 和 echobuf}
begin
    local x
    P(T_Ready)
    从键盘数据传输缓冲 x 中取出字符 m,记为 x.m
    if 为控制消息
        P(inbuffer)
    send(x.m)
    将 x.m 送入 echobuf
    V(T_Busy)
end
```

键盘控制进程 KCP：

```
repeat
    P(T_Busy)
    把输入字符放入数据传输缓冲
    V(T_Ready)
until 终端关闭
```

显示器控制进程 DCP：

设 D_Ready 和 D_Busy 分别为 DP 和 DCP 的私有信号量,且初始值为 0 和 1。

```
初始化{清除输出缓冲 outbuf,echo 模式置 false}
begin
    if outbuf 满
    then
        receive(k)
        P(D_Busy)
        把 k 送入显示器数据缓冲区
        V(D_Ready)
    else
        echo 模式置 true
        echo buf 中字符置入显示器数据缓冲区 fi
end
```

显示器动作 DP：

```
repeat
    if echo 模式
    then
        打印显示器数据缓冲区中的字符
    else
        P(D_Ready)
        打印显示器数据缓冲区中的消息
        V(D_Busy)
until 显示器关机
```

设过程 read(x)把 inbuf 中的所有字符读到用户进程数据区 x 处，过程 write(y)把用户进程 y 处的消息写到 outbuf 中。read(x)和 write(y)的描述略。

3.13 编写一个程序使用系统调用 fork 生成 3 个子进程，并使用系统调用 pipe 创建一个管道，使得这 3 个子进程和父进程公用同一管道进行信息通信。

答：

```
main()
{
    int r,p1,p2,p3,fd[2];                    /* fd[2]为管道文件读写标识 */
    char buf[50],s[5];
    pipe(fd);                                /* 创建管道 pipe() */
    while((p1=fork())==-1);                  /* 创建子进程 1 */
    if(p1==0)                                /* 在子进程 1 中执行 */
    {
        lockf(fd[1],1,0);                    /* 锁定写过程 */
        sprintf(buf,"child process P1 is sending message!\n");
        printf("child process P1!\n");
        write(fd[1],buf,50);                 /* 将 buf 中的数据写入 pipe */
        sleep(5);                            /* 睡眠等待父进程读出 */
        lockf(fd[1],0,0);                    /* 解锁 */
        exit(0);
    }
    else
    {
        while((p2=fork())==-1);              /* 创建进程 2 */
        if(p2==0)                            /* 在子进程 2 中执行 */
        {
            lockf(fd[1],1,0);                /* 锁定写过程 */
            sprintf(buf, "child process P2 is sending message!\n");
                                             /* 数据写入 buf */
            printf("child process P2!\n");
            write(fd[1],buf,50);             /* 将 buf 中的数据写入 pipe */
            sleep(5);                        /* 同步等待父进程读 */
            lockf(fd[1],0,0);                /* 解锁 */
            exit(0);                         /* 释放进程资源 */
        }
        else
        {
            while((p3=fork())==-1);          /* 创建进程 3 */
            if(p3==0)                        /* 在子进程 3 中执行 */
            {
                lock(fd[1],1,0);
                sprintf(buf,"child process P3 is sending message!\n");
                printf("child process P3!\n");
```

```
            write(fd[1],buf,50);
            sleep(5);
            lockf(fd[1],0,0);
            exit(0);
        }
        wait(0);                        /* 父进程等待子进程先执行 */
        if(r= read(fd[0],s,50)= = -1)   /* 读管道 pipe 内容到 s 中 */
            printf("can't read pipe\n");
        else
            printf("% s\n",s);
        wait(0);                        /* 等待另一个子进程执行 */
        if(r= read(fd[0],s,50)= = -1)
            printf("can't read pipe\n");
        else
            printf("% s\n",s);
        wait(0);                        /* 等待最后一个子进程执行 */
        if(r= read(fd[0],s,50)= = -1)
            printf("can't read pipe\n")
        else
            printf("% s\n",s);
        exit(0);
    }
  }
}
```

3.14 设有 5 个哲学家,共享一张放有 5 把椅子的桌子,每人分得一把椅子。但是,桌子上总共只有 5 支筷子,在每人两边分开各放一支。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。条件如下:

(1) 只有拿到两支筷子时,哲学家才能吃饭。

(2) 如果筷子已在他人手上,则该哲学家必须等到他人吃完之后才能拿到筷子。

(3) 任一哲学家在自己未拿到两支筷子吃饭之前,决不放下自己手中的筷子。

试解答以下问题:

(1) 描述一个保证不会出现两个邻座同时要求吃饭的通信算法。

(2) 描述一个既没有两邻座同时吃饭,又没有人饿死(永远拿不到筷子)的算法。在什么情况下,5 个哲学家全部吃不上饭?

答:

(1) 设信号量 c[0]~c[4],初始值均为 1,分别表示 i 号筷子被拿(i=0,1,2,3,4)。

```
send(i): 第 i 个哲学家要吃饭
begin
    P(c[i]);
    P(c[i+1 mod 5]);
    eat;
    V(c[i+1 mod 5]);
```

```
        V(c[i]);
    end
```

该过程能保证两邻座不同时吃饭，但会出现 5 个哲学家一人拿一支筷子，谁也吃不上饭的死锁情况。

（2）解决的思路如下：让奇数号的哲学家先取右手边的筷子，让偶数号的哲学家先取左手边的筷子。

这样，任何一个哲学家拿到一支筷子以后，就已经阻止了他邻座的一个哲学家吃饭的企图，除非某个哲学家一直吃下去，否则不会有人会饿死。

```
    send(i):
    begin
        if i mod 2==0
        then
        {
            P(c[i]);
            P(c[i+1] mod 5);
            eat;
            V(c[i]);
            V(c[i+1 mod 5])
        }
        else
        {
            P(c[i+1 mod 5]);
            P(c[i]);
            eat;
            V(c[i+1 mod 5]);
            V(c[i]);
        }
    end
```

3.15　什么是线程？试述线程与进程的区别。

答：线程是在进程内用于调度和占有处理机的基本单位，它由线程控制表、存储线程上下文的用户栈以及核心栈组成。线程可分为用户级线程、核心级线程以及用户/核心混合型线程等类型。其中用户级线程在用户态下执行，CPU 调度算法和各线程优先级都由用户设置，与操作系统内核无关。核心级线程的调度算法及线程优先级的控制权在操作系统内核。混合型线程的控制权则在用户和操作系统内核二者。

线程与进程的主要区别如下：

（1）进程是资源管理的基本单位，它拥有自己的地址空间和各种资源，例如内存空间、外部设备等；线程只是处理机调度的基本单位，它只和其他线程一起共享进程资源，但自己没有任何资源。

（2）以进程为单位进行处理机切换和调度时，由于涉及资源转移以及现场保护等问题，将导致处理机切换时间变长，资源利用率降低。以线程为单位进行处理机切换和调度时，由

于不发生资源变化，特别是地址空间的变化，处理机切换的时间较短，从而处理机效率较高。

(3) 对用户来说，多线程可减少用户的等待时间，提高系统的响应速度。例如，当一个进程需要对两个不同的服务器进行远程过程调用时，对于无线程系统的操作系统来说，需要顺序等待两个不同调用返回结果后才能继续执行，且在等待中容易发生进程调度。对于多线程系统而言，则可以在同一进程中使用不同的线程同时进行远程过程调用，从而缩短进程的等待时间。

(4) 线程和进程一样，都有自己的状态，也有相应的同步机制。不过，由于线程没有单独的数据和程序空间，因此，线程不能像进程的数据与程序那样交换到外存存储空间，从而线程没有挂起状态。

(5) 进程的调度、同步等控制大多由操作系统内核完成，而线程的控制既可以由操作系统内核进行，也可以由用户控制进行。

3.16 使用库函数 clone()与 pthread_create()，在 Linux 环境下创建两种不同执行模式的线程程序。

答： Linux 系统支持用户级线程和核心级线程两种执行模式，其库函数分别为 pthread_create() 和 clone()。创建用户级线程和核心级线程的程序示例如下。

(1) 用户级线程编程示例：

```
#include <pthread.h>
void * ptest(void * arg)
{
    printf("This is the new thread!\n");
    return(NULL);
}

main()
{
    pthread_t tid;
    printf("This is the parent process!\n");
    pthread_create(&tid,NULL,ptest,NULL);     /* 创建线程 */
    sleep(1);
    return;
}
```

该程序通过调用 pthread_create()创建一个用户级线程，其指针为 tid，过程名为 ptest。

(2) 核心级线程编程示例：

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/unistd.h>

#define STACKSIZE 16384
```

```c
#define CSIGNAL 0x000000ff            /* signal mask to be sent at exit */
#define CLONE_VM 0x00000100           /* set if VM shared between processes */
#define CLONE_FS 0x00000200           /* set if info shared between processes */
#define CLONE_FILES 0x00000400        /* set if files shared between processes */
#define CLONE_SIGHAND 0x00000800      /* set if signal handlers shared between
                                         processes */

int show_same_vm;

void cloned_process_start_here(void * data)
{
    printf("child:\t got argument %d as fd\n", (int) data);
    show_same_vm= 5;
    printf ("child: \t vm= %d \n", show_same_vm);
    close((int)data);
}

int main()
{
    int fd, pid;

    fd= open (" /dev/null", O_RDWR);
    if (fd < 0)
    {
        perror("/dev/null");
        exit(1);
    }
    printf ("mother: \t vm= %d\n",fd);

    show_same_vm= 10;
    printf ("mother: \t vm= %d\n", show_same_vm);

    pid= clone (cloned_process_start_here, (void * ) fd);
    if (pid < 0)
    {
        perror ("start_thread");
        exit(1);
    }
    sleep(1);
    printf ("mother: \t vm= %d\n", show_same_vm);
    if (write (fd, "c", 1) < 0)
        printf ("mother: \t child closed our file descriptor\n");
}

#include <signal.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/unistd.h>

#define STACKSIZE 16384
#define CSIGNAL 0x000000ff                /* signal mask to be sent at exit */
#define CLONE_VM 0x00000100               /* set if VM shared between processes */
#define CLONE_FS 0x00000200               /* set if info shared between processes */
#define CLONE_FILES 0x00000400            /* set if files shared between processes */
#define CLONE_SIGHAND 0x00000800                  /* set if signal handlers shared between processes */

int clone (void (* fn) (void * ), void * data)       /* 创建核心线程 */
{
        long retval,errno;
        void **newstack;
        /*
         * allocate new stack for subthread
         */
newstack= (void **) malloc (STACKSIZE);
if (!newstack)
        return -1;
/*
 * Set up the stack for child function, put the (void * )
 * argument on the stack
 */
newstack= (void **) (STACKSIZE+ (char * ) newstack);
*newstack= data;

/*
 * Do clone () system call. We need to do the low-level stuff
 * entirely in assembly as we're returning with a different
 * stack in the child process and we couldn't otherwise guarantee
 * that the program doesn't use the old stack incorrectly.
 *
 * Parameters to clone() system call:
 * %eax-_NR_clone, clone system call number
 * %ebx-clone_flags, bitmap of cloned data
 * %ecx-new stack pointer for cloned child
 *
 * In this example %ebx is CLONE_VM | CLONE_FS | CLONE_FILES |
 * CLONE_SIGHAND which shares as much as possible between parent and
 * child. (the signal to be sent on child termination into clone_flags:
 * SIGCHLD makes the cloned process work like a "normal" UNIX child
 * process)
```

```
 *
 * The clone () system call returns (in %eax) the pid of the newly
 * cloned process to the parent, and 0 to the cloned process. If
 * an error occurs, the return value will be the negative errno.
 * In the child process, we will do a "jsr" to the requested function
 * and then do a "exit() " system call which will terminate the child.
 */
_asm_volatile_(
    "int $0x80\n\t"              /* Linux/i386 system call */
    "testl %0, %0\n\t"           /* check return value */
    "jne lf\n\t"                 /* jump if parent */
    "call *%3\n\t"               /* start subthread function */
    "movl %2, %0\n\t"
    "int $0x80\n"                /* exit system call: exit subthread */
    "l: \t"
    : "=a" (retval)
    : "0" (_NR_clone), "i" (_NR_exit),
    "r" (fn),
    "b" (CLONE_VM | CLONE_FS | CLONE_FILES |
    CLONE_SIGHAND | CLONE_SIGCHLD),
    "c" (newstack));

    if (retval < 0)
    {
        errno= - retval;
        retval= - 1;
    }
    return retval;
}
```

第 4 章　处理机调度

4.1　什么是分级调度？分时系统中有作业调度的概念吗？如果没有，为什么？

答：处理机调度问题实际上也是处理机的分配问题。显然只有那些参与竞争处理及所必需的资源都已得到满足的进程才能享有竞争处理机的资格。这时它们处于内存就绪状态。这些必需的资源包括内存、外设及有关数据结构等。从而，在进程有资格竞争处理机之前，作业调度程序必须先调用存储管理和外设管理程序，并按一定的选择顺序和策略从输入井中选择出几个处于后备状态的作业，为它们分配资源和创建进程，使它们获得竞争处理机的资格。另外，由于处于执行状态下的作业一般包括多个进程，而在单机系统中，每一时刻只能有一个进程占有处理机，这样，在外存中，除了处于后备状态的作业外，还存在处于就绪状态而等待得到内存的作业。需要有一定的方法和策略为这部分作业分配空间。因此处理机调度需要分级。

一般来说，处理机调度可分为 4 级：

(1) 作业调度。又称宏观调度或高级调度。

(2) 交换调度。又称中级调度。其主要任务是按照给定的原则和策略，将处于外存交换区中的就绪状态或等待状态或内存等待状态的进程交换到外存交换区。交换调度主要涉及内存管理与扩充，因此在有些书本中也把它归入内存管理部分。

(3) 进程调度。又称微观调度或低级调度。其主要任务是按照某种策略和方法选取一个处于就绪状态的进程占用处理机。在确立了占用处理机的进程之后，系统必须进行进程上下文切换以建立与占用处理机进程相适应的执行环境。

(4) 线程调度。进程中相关堆栈和控制表等的调度。

在分时系统中，一般不存在作业调度，而只有线程调度、进程调度和交换调度。这是因为在分时系统中，为了缩短响应时间，作业不是建立在外存中，而是直接建立在内存中。在分时系统中，一旦用户和系统的交互开始，用户马上要进行控制。因此，分时系统中没有作业提交状态和后备状态。分时系统的输入信息经过终端缓冲区为系统直接接收，或立即处理，或经交换调度暂存外存中。

4.2　试述作业调度的主要功能。

答：作业调度的主要功能是：按一定的原则对外存输入井中的大量后备作业进行选择，给选出的作业分配内存、输入输出设备等必要的资源，并建立相应进程，使该作业的相关进程获得竞争处理机的权利。另外，当作业执行完毕时，还负责回收系统资源。

4.3　作业调度的性能评价标准有哪些？这些性能评价标准在任何情况下都能反映调度策略的优劣吗？

答：对于批处理系统，由于主要用于计算，因而对于作业的周转时间要求较高，从而作业的平均周转时间或平均带权周转时间被用来衡量调度程序的优劣。但对于分时系统来

说，平均响应时间又被用来衡量调度策略的优劣。

对于分时系统，除了要保证系统吞吐量大、资源利用率高之外，还应保证用户能够容忍的响应时间。因此，在分时系统中，仅仅用周转时间或带权周转时间来衡量调度性能是不够的。

对于实时系统来说，衡量调度算法优劣的主要标志则是满足用户要求的时限时间。

4.4　进程调度的功能有哪些？

答：进程调度的功能如下：

(1) 记录和保存系统中所有进程的执行情况。

(2) 选择占有处理机的进程。

(3) 进行进程上下文切换。

4.5　进程调度的时机有哪几种？

答：进程调度的时机有：

(1) 正在执行的进程执行完毕。这时如果不选择新的就绪进程执行，将浪费处理机资源。

(2) 执行中进程自己调用阻塞原语将自己阻塞起来进入睡眠等待状态。

(3) 执行中进程调用了 P 原语操作，从而因资源不足而被阻塞；或调用了 V 原语操作激活了等待资源的进程队列。

(4) 执行中进程提出 I/O 请求后被阻塞。

(5) 在分时系统中时间片已经用完。

(6) 在执行完系统调用等系统程序后返回用户程序时，可看做系统进程执行完毕，从而调度选择一个新的用户进程执行。

在 CPU 执行方式是可剥夺时，还有：

(7) 就绪队列中的某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

4.6　假设有 4 道作业，它们的提交时间及执行时间由下表给出。

作业号	提交时刻/hh:mm	执行时间/hr
1	10:00	2
2	10:20	1
3	10:40	0.5
4	10:50	0.3

计算在单道程序环境下，采用先来先服务调度算法和最短作业优先调度算法时的平均周转时间和平均带权周转时间，并指出它们的调度顺序。

答：(1) 先来先服务调度顺序如下：

① $Ts_1=10:00$　$Te_1=12:00$　$T_1=2.00$　$Tw_1=0$

② $Ts_2=10:20$　$Te_2=13:00$　$T_2=1.00$　$Tw_2\approx 1.70$

③ Ts_3＝10:40　Te_3＝13:30　T_3＝0.50　Tw_3≈2.30
④ Ts_4＝10:50　Te_4＝13:48　T_4＝0.30　Tw_4≈2.70
T＝0.25＊(2＋2.7＋2.8＋3)＝2.625
W＝0.25＊(4＋0＋1.7/1＋2.3/0.5＋2.7/0.3)＝4.825
(2) 最短作业优先调度顺序如下：
① Ts_4＝10:50　Te_4＝11:08　T_4＝0.3　Tw_4＝0
② Ts_3＝10:40　Te_3＝11:28　T_3＝0.5　Tw_3＝0.3
③ Ts_2＝10:20　Te_2＝12:08　T_2＝1　Tw_2＝0.8
④ Ts_1＝10:00　Te_1＝13:48　T_1＝2　Tw_1＝1.8
T＝0.25＊(0.3＋0.8＋1.8＋3.8)＝1.675
W＝0.25＊(4＋0＋0.30/0.50＋0.8/1＋1.8/2)＝1.575

4.7　设某进程所需要的服务时间 $t=k*q$，其中，k 为时间片的个数，q 为时间片长度且为常数。当 t 为一定值时，令 q 趋于 0，则有 k 趋于无穷，从而服务时间为 t 的进程响应时间 T 为 t 的连续函数。对应于时间片的轮转调度方式(RR)、先来先服务方式(FCFS)和线性优先级调度方式(SRR)，其响应时间函数分别为：

$$T_{rr}(t)=t*\mu/(\mu-\lambda)$$
$$T_{fc}(t)=1/(\mu-\lambda)$$
$$T_{sr}(t)=1/(\mu-\lambda)-(1-t*\mu)/(\mu-\lambda')$$

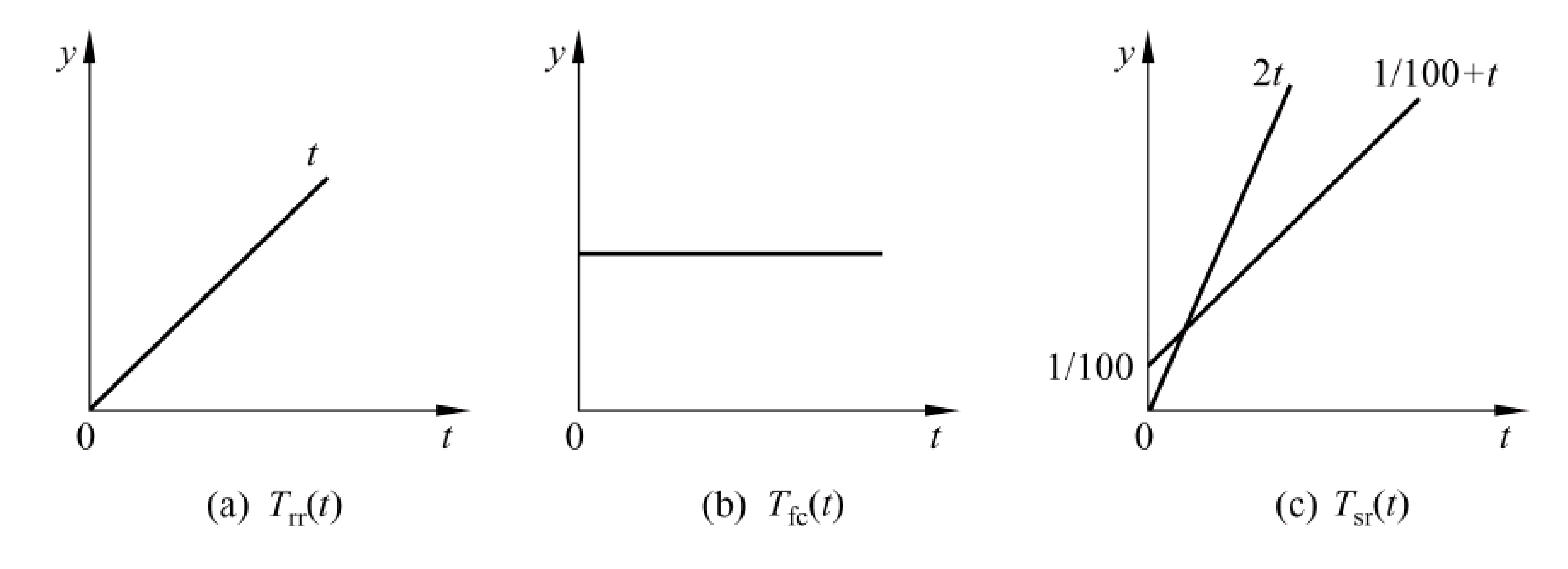
其中 $\lambda'=(1-b/a)*\lambda=r*\lambda$。

取 $(\lambda,\mu)=(50,100)$ 和 $(\lambda,\mu)=(80,100)$，分别改变 r 的值，画出 $T_{rr}(t)$、$T_{fc}(t)$ 和 $T_{sr}(t)$ 的时间变化图。

答：(1) 对 $(\lambda,\mu)=(50,100)$，有

$$T_{rr}(t)=t,\ T_{fc}(t)=t/50,\ T_{sr}(t)=1/50-(1-100t)/(100-50r)$$

$r\to 0$ 时，$T_{sr}(t)=1/100+t$；
$r\to 1$ 时，$T_{sr}(t)=2t$。
时间变化图如下图所示。



(a) $T_{rr}(t)$　(b) $T_{fc}(t)$　(c) $T_{sr}(t)$

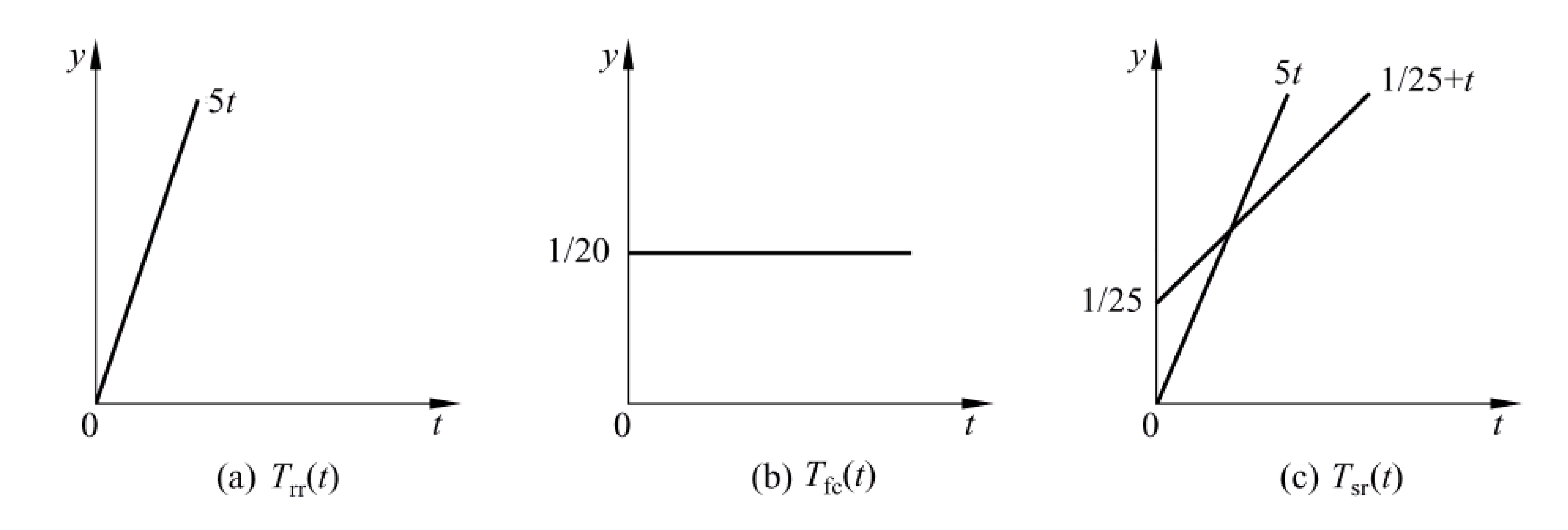
只有 $T_{sr}(t)$ 受 r 值影响。且 r 增大时，$T_{sr}(t)$ 斜率增大，服务时间也增加。
(2) 对 $(\lambda,\mu)=(80,100)$，有

$$T_{rr}(t)=5t,\ T_{fc}(t)=1/20,\ T_{sr}(t)=1/20-(1-100t)/(100-80r)$$

$r\to 0$ 时，$T_{sr}(t)\to 1/25+t$；

$r \to 1$ 时，$T_{sr}(t) \to 5t$。

时间变化图如下图所示。



$T_{sr}(t)$的斜率随 r 增大而增大，y 截距由 1/25 到 0 逐渐移动。

4.8　什么是多处理机系统？并行处理系统、计算机网络、分布式系统和多处理机系统的操作系统之间有何区别？

答：从广义上说，使用多台处理机协调工作来完成用户所要求任务的计算机系统都是多处理机系统。狭义的多处理机系统是利用系统内的多个 CPU 来并行执行用户的几个程序，以提高系统的吞吐量；或用来进行冗余操作，以提高系统的可靠性。

并行处理机系统是利用多个功能单元（CPU）执行同一程序，多个处理机在物理位置上处于同一块电路板上。计算机网络系统则是通过物理通信媒介，包括有线和无线的，把现有的分散的计算机系统互相连接起来，以达到信息传递和资源共享的目的。分布式系统是以计算机网络为基础的，对用户来说是透明的。多处理机系统是指在同一计算机系统内共享内存的计算机系统。

4.9　什么是实时调度？它与非实时调度有何区别？

答：实时调度是为了完成实时处理任务而分配计算机处理器的调度方法。

实时处理任务要求计算机在用户允许的时限范围内给出计算机响应信号。实时处理任务可分为硬实时任务（hard real-time task）和软实时任务（soft real-time task）。硬实时任务要求计算机系统必须在用户给定的时限内处理完毕，软实时任务允许计算机系统在用户给定的时限左右处理完毕。

针对硬实时任务和软实时任务，计算机系统可以有不同的实时调度算法。这些算法采用基于优先级的抢先式调度策略，具体地说，大致有如下几类：

（1）静态表驱动模式。该模式用于周期性实时调度，它在任务到达之前对各任务抢占处理机的时间进行分析，并根据分析结果进行调度。

（2）静态优先级驱动的抢先式调度模式。该模式也进行静态分析。分析结果不是用于调度，只是用于给各任务指定优先级。系统根据各任务的优先级进行抢先式调度。

（3）基于计划的动态模式。该模式在新任务到达后，将以前调度过的任务与新到达的任务一起统一计划，分配 CPU 时间。

（4）动态尽力而为模式。该模式不进行任何关于资源利用率的分析，只检查各任务的时限是否能得到满足。

代表性的实时调度算法有两种，即时限式调度法(deadline scheduling)和频率单调调度法(rate monotonic scheduling)。

实时调度与非实时调度的主要区别如下：

(1) 实时调度所调度的任务有完成时限，而非实时调度没有。因此，实时调度算法的正确与否不仅与算法的逻辑有关，也与调度算法调度的时限有关。

(2) 实时调度要求较短的进程或线程切换时间，而非实时调度的进程或线程的切换时间较长。

(3) 非实时调度强调资源利用率(批处理系统)或用户共享处理机(分时系统)，实时调度则主要强调在规定时限范围内完成对相应设备的控制。

(4) 实时调度为抢先式调度，而非实时调度则很少采用抢先式调度。

4.10 写出图 4.11 所示周期性任务调度用的时限调度算法。

答：首先设置周期性任务进程的数据结构：

```
process struct
{
      int p_num;                  /* 进程号 */
      int arr_time;               /* 进程到达时间 */
      int exe_time;               /* 进程所需执行时间 */
      int end_deadline;           /* 时限 */
      int period;                 /* 周期 */
      int passed_time;            /* 该进程已占有处理机时间 */
} pro[N],pro1[n];
local int N,n,T1,T2;              /* N,n 为正整数,T1,T2 为进程周期 */
```

算法描述如下：

```
begin
      if {n 个进程 pro1[n]到达,且要求调度}
      then {
            初始化 pro1[n]
            和 T1,T2 分别比较 pro1[n].period
            if {pro1[n].period ≠ T1 且 pro1[n].period ≠ T2}
            then {返回错误信息:该进程周期错误}
            else {   比较 pro1[n]与 pro[N]中的各进程时限
                     选择 pro1[n].end_deadline 或 pro[N].end_deadline 中时间最近者占据
                     处理机
                     保护当前进程现场,并将其放入 pro[N]队列
                     将未选中的 pro1[n]的其他进程置入 pro[N]
                 }
      }
      else
            if 当前进程执行完毕要求调度
            {
```

```
            比较 pro[N]中每个进程的 end_deadline
            选择 end_deadline最小的进程占有处理机
        }
        else
            等待占有处理机的进程执行完毕
end
```

4.11 设周期性任务 P_1、P_2、P_3的周期 T_1、T_2、T_3分别为 100、150、350；执行时间分别为 20、40、100。问：是否可用频率单调调度算法进行调度？

答：根据频率单调调度公式，能进行周期性调度的进程应满足下式：

$$c_1/T_1+c_2/T_2+\cdots+c_n/T_n\leqslant n(2^{1/n}-1)$$

在上式中，$n=3$，$T_1=100$，$T_2=150$，$T_3=350$，而 c_1、c_2、c_3 分别等于 20、40、100，从而有

$$20/100+40/150+100/350=0.2+0.267+0.286=0.753 \qquad (1)$$

而

$$3\times(2^{1/3}-1)=0.779 \qquad (2)$$

比较式(1)与式(2)，有 (1)≤(2)。因此，本题给出的周期性任务可以用频率单调调度算法进行调度。

第5章　存储管理

5.1　存储管理的主要功能是什么？

答：存储管理的主要功能包括以下几点：

（1）在硬件的支持下完成统一管理内存和外存之间数据和程序段自动交换的虚拟存储器功能。

（2）将多个虚存的一维线性空间或多维线性空间变换到内存的唯一的一维物理线性地址空间。

（3）控制内外存之间的数据传输。

（4）实现内存的分配和回收。

（5）实现内存信息的共享与保护。

5.2　什么是虚拟存储器？其特点是什么？

答：由进程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟存储器。虚拟存储器不考虑物理存储器的大小和信息存放的实际位置，只规定每个进程中相互关联的信息的相对位置。每个进程都拥有自己的虚拟存储器，虚拟存储器的容量由计算机的地址结构和寻址方式来确定。

实现虚拟存储器要求有相应的地址转换机构，以便把指令的虚拟地址变换为实际物理地址；另外，由于内存空间较小，进程只有部分内容存放于内存中，待执行时根据需要再调指令入内存。

5.3　实现地址重定位的方法有哪几类？

答：实现地址重定位的方法有两类：静态地址重定位和动态地址重定位。

（1）静态地址重定位是在虚拟空间程序执行之前由装配程序完成地址映射工作。静态重定位的优点是不需要硬件支持，但是用静态地址重定位方法进行地址变换无法实现虚拟存储器。静态地址重定位的另一个缺点是必须占用连续的内存空间和难以做到程序和数据的共享。

（2）动态地址重定位是在程序执行过程中，在CPU访问内存之前由硬件地址变换机构将要访问的程序或数据地址转换成内存地址。动态地址重定位的主要优点如下：

① 可以对内存进行非连续分配。

② 动态重定位提供了实现虚拟存储器的基础。

③ 动态重定位有利于程序段的共享。

形式化描述：略。

5.4　常用的内存信息保护方法有哪几种？它们各自的特点是什么？

答：常用的内存信息保护方法有硬件法、软件法和软硬件结合保护法3种。

上下界保护法是一种常用的硬件保护法。上下界存储保护技术要求为每个进程设置一对上下界寄存器。上下界寄存器中装有被保护程序和数据段的起始地址和终止地址。在程序执行过程中,在对内存进行访问操作时首先进行访问地址合法性检查,即检查经过重定位之后的内存地址是否在上下界寄存器所规定的范围之内。若在规定的范围之内,则访问是合法的;否则是非法的,并产生访问越界中断。

保护键法也是一种常用的软件存储保护法。保护键法为每一个被保护存储块分配一个单独的保护键。在程序状态字中则设置相应的保护键开关字段,对不同的进程赋予不同的开关代码以和被保护的存储块中的保护键匹配。保护键可以设置成对读写同时保护的或只对读写进行单项保护的。如果开关字段与保护键匹配或者存储块未受到保护,则访问该存储块是允许的,否则将产生访问出错中断。

另外一种常用的硬软件内存保护方式是界限存储器与 CPU 的用户态、核心态相结合的保护方式。在这种保护方式下,用户态进程只能访问那些在界限寄存器所规定范围内的内存部分,而核心态进程则可以访问整个内存地址空间。

5.5 如果把 DOS 的执行模式改为保护模式,起码应做怎样的修改?

答:如果要把 DOS 的执行模式改成保护模式,起码要为每一个进程设置一对上下界寄存器。上下界寄存器中装有被保护程序和数据段的起始地址和终止地址。在程序执行过程中,在对内存进行访问操作时首先进行访问地址合法性检查,即检查经过重定位之后的内存地址是否在上下界寄存器所规定的范围之内。若在规定的范围之内,则访问是合法的;否则是非法的,并产生访问越界中断。另外,还应该把指令的访问内存模式由访问物理地址改为由逻辑地址变换为物理地址的方式。

5.6 动态分区式管理的常用内存分配算法有哪几种?比较它们各自的优缺点。

答:动态分区式管理的常用内存分配算法有最先适应法(FF)、最佳适应法(BF)和最坏适应法(WF)。

这几种算法的优缺点比较如下:

(1) 从搜索速度上看,最先适应法最佳,最佳适应法和最坏适应法都要求把空闲区按大小进行排队。

(2) 从回收过程来看,最先适应法也最佳,因为最佳适应法和最坏适应法都必须重新调整空闲区的位置。

(3) 最佳适应法找到的空闲区是最佳的,但是会造成内存碎片较多,影响了内存利用率;而最坏适应法的内存碎片最少,但是对内存的请求较多的进程有可能分配失败。

总之,3 种算法各有所长,针对不同的请求队列,它们的效率和功能是不一样的。

5.7 5.3 节讨论的分区式管理可以实现虚存吗?如果不能,需要怎样修改?试设计一个分区式管理实现虚存的程序流程图。如果能,试说明理由。

答:5.3 节讨论的分区式管理不能实现虚存。如果要实现虚存,可以在分区的基础之上对每个分区内部进行请求调页式管理。

程序流程图:略。

5.8　什么是覆盖？什么是交换？覆盖和交换的区别是什么？

答：将程序划分为若干个功能上相对独立的程序段，按照程序的逻辑结构让那些不会同时执行的程序段共享同一块内存区的内存扩充技术就是覆盖。

交换是指先将内存某部分的程序或数据写入外存交换区，再从外存交换区中调入指定的程序或数据到内存中来，并让其执行的一种内存扩充技术。

与覆盖技术相比，交换不要求程序员给出程序段之间的覆盖结构，而且，交换主要在进程或作业之间进行，而覆盖则主要在同一个作业或同一个进程内进行。另外，只能对那些与覆盖程序段无关的程序段进行覆盖。

5.9　什么是页式管理？静态页式管理可以实现虚存吗？

答：页式管理就是把各进程的虚拟空间划分为若干长度相等的页面，把指令按页面大小划分后存放在内存中执行，或者只在内存中存放那些经常被执行或即将被执行的页面，而那些不被经常执行以及在近期内不可能被执行的页面则存放于外存中，按一定规则调入的一种内存管理方式。

静态页式管理不能实现虚存，这是因为静态页式管理要求进程或作业在执行前全部被装入内存，作业或进程的大小仍受内存可用页面数的限制。

5.10　什么是请求页式管理？试设计和描述一个请求页式管理时的内存页面分配和回收算法（包括缺页处理部分）。

答：请求页式管理是动态页式内存管理的一种，它在作业或进程开始执行之前，不把作业或进程的程序段和数据段一次性全部装入内存，而只装入被认为是经常反复执行和调用的工作区部分。其他部分则在执行过程中动态装入。请求页式管理的调入方式是，当需要执行某条指令而又发现它不在内存时，或当执行某条指令需要访问其他数据或指令，而这些指令和数据又不在内存中，从而发生缺页中断时，系统将外存中相应的页面调入内存。

请求页式管理的内存页面分配和回收算法：略。

5.11　请求页式管理中有哪几种常用的页面置换算法？试比较它们的优缺点。

答：比较常用的页面置换算法有：

（1）随机淘汰算法（random glongram）。即随机地选择某个用户页面并将其换出。

（2）轮转法（Round Robin，RR）。轮转法循环换出内存可用区内一个可以被换出的页，无论该页是刚被换进或已经换进内存很长时间。

（3）先进先出法（First In First Out，FIFO）。FIFO 算法选择在内存驻留时间最长的一页将其淘汰。

（4）最近最久未使用页面置换算法（Least Recently Unused，LRU）。该算法的基本思想是：当需要淘汰某一页时，选择离当前时间最近的一段时间内最久没有使用过的页面先淘汰。

（5）理想型淘汰算法（Optimal Replacement Algorithm，OPT）。该算法淘汰在访问串中将来再也不出现的或是在离当前最远的位置上出现的页面。

RR 和 FIFO 都是基于 CPU 按线性顺序访问地址空间这一假设，但是实际上 CPU 在很多时候并非是按线性顺序访问地址空间的，因而其内存利用率不高。此外 FIFO 算法还存在着 Belady 现象。LRU 算法的完全实现是相当困难的，因此在实际系统中往往要采取 LRU 的近似算法，常用的近似算法有最不经常使用页面淘汰算法（Least Frequently Used，LFU）和最近没有使用页面淘汰算法（NUR）。OPT 算法由于必须预先知道每一个进程的指令访问串，所以它是无法实现的。

5.12　什么是 Belady 现象？试找出一个 Belady 现象的例子。

答：Belady 现象是指在使用 FIFO 算法进行内存页面置换时，在未给进程或作业分配足够它所要求的全部页面的情况下，有时出现的分配的页面数增多，缺页次数反而增加的奇怪现象。

例：假设进程 P 共有 5 页，程序访问内存的顺序（访问串）为 1，2，3，4，1，2，5，1，2，3，4，5。

当内存工作区页面为 3 时：

页面数＝3　缺页次数＝9　缺页率＝9/12＝75％

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
√	√	√	√	√	√	√			√	√	

当内存工作区页面为 4 时：

页面数＝4　缺页次数＝10　缺页率＝10 / 12＝83.3％

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
√	√	√	√			√	√	√	√	√	√

由上例可知，在 FIFO 算法产生了 Belady 现象时，工作区页面增加反而使得缺页率变大。

5.13　描述一个包括页面分配与回收、页面置换和存储保护的请求式存储管理系统。

答：略。

5.14　什么是段式管理？它与页式管理有何区别？

答：段式管理就是将程序按照内容或过程（函数）关系分成段，每段拥有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间，也就是一个二维虚拟存储器。段式管理程序以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换成实际的内存物理地址。同页式管理时一样，段式管理也采用只把那些经常访问的段驻留内存，而把那些在将来一段时间内不被访问的段放入外存，待需要时自动调入相关段的方法实现二维虚拟存储器。

段式管理和页式管理的主要区别如下：

（1）页式管理中源程序进行编译链接时是将主程序、子程序、数据区等按照线性空间的一维地址顺序排列起来。段式管理则是将程序按照内容或过程（函数）关系分成段，每段拥有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间，也就是一个二维虚拟存储器。

（2）同动态页式管理一样，段式管理也提供了内外存统一管理的虚存实现。与页式管理不同的是：段式虚存每次交换的是一段有意义的信息，而不是像页式虚存管理那样只交换固定大小的页，从而需要多次的缺页中断才能把所需信息完整地调入内存。

（3）在段式管理中，段长可根据需要动态增长。这对那些需要不断增加或改变新数据或子程序的段来说，将是非常有好处的。

（4）段式管理便于对具有完整逻辑功能的信息段进行共享。

（5）段式管理便于进行动态链接，而页式管理进行动态链接的过程非常复杂。

5.15　段式管理可以实现虚存吗？如果可以，简述实现方法。

答：段式管理可以实现虚存。

段式管理把程序按照内容或过程（函数）关系分成段，每段拥有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间（段号 s 与段内相对地址 w），也就是一个二维虚拟存储器。段式管理以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换成实际的内存物理地址。只把那些经常访问的段驻留内存，而把那些在将来一段时间内不被访问的段放入外存，待需要时产生缺段中断，自动调入。

5.16　为什么要提出段页式管理？它与段式管理及页式管理有何区别？

答：因为段式管理和页式管理各有所长。段式管理为用户提供了一个二维的虚拟地址空间，反映了程序的逻辑结构，有利于段的动态增长以及共享和内存保护等，这极大地方便了用户。而分页系统则有效地克服了碎片，提高了存储器的利用效率。从存储管理的目的来讲，主要是方便用户的程序设计和提高内存的利用率。所以人们提出了将段式管理和页式管理结合起来让其互相取长补短的段页式管理。

段页式管理与段式和页式管理相比，其访问时间较长，因此执行效率低。

5.17　为什么说段页式管理时的虚拟地址仍是二维的？

答：因为在段页式内存管理中，对每一段内的地址空间进行分页式管理只是为了克服在内存分配过程中产生的大量碎片，从而提高存储器的利用效率，它并没有改变段内地址空

间的一维结构，所以段页式内存管理中的虚拟地址仍然和段式内存管理中的虚拟地址一样，是二维结构的。

5.18　段页式管理的主要缺点是什么？有什么改进办法？

答：段页式管理的主要缺点是对内存中指令或数据进行存取时，至少需要对内存进行三次以上的访问。第一次是由段表地址寄存器取段表始址后访问段表，由此取出对应段的页表在内存中的地址。第二次则是访问页表得到所要访问的指令或数据的物理地址。只有在访问了段表和页表之后，第三次才能访问真正需要访问的物理单元。显然，这将大大降低CPU 执行指令的速度。

改进办法是设置快速联想寄存器。在快速联想寄存器中，存放当前最常用的段号 s、页号 p 和对应的内存页面地址与其他控制项。当需要访问内存空间某一单元时，可在通过段表、页表进行内存地址查找的同时，根据快速联想寄存器查找其段号和页号。如果所要访问的段或页的地址在快速联想寄存器中，则系统不再访问内存中的段表、页表，而直接把快速联想寄存器中的值与页内相对地址 d 拼接起来得到内存地址。

5.19　什么是局部性原理？什么是抖动？你有什么办法减少系统的抖动现象？

答：局部性原理是指在几乎所有程序的执行过程中，在一段时间内，CPU 总是集中地访问程序中的某一个部分而不是对程序的所有部分具有平均的访问概率。

抖动是指当给进程分配的内存小于所要求的工作区时，由于内存和外存之间交换频繁，访问外存的时间和输入输出处理时间大大增加，造成 CPU 因等待数据而空转，使得整个系统性能大大下降。

在物理系统中，为了防止抖动的产生，在进行淘汰或置换时，一般总是把缺页进程锁住，不让其换出，从而防止抖动发生。

防止抖动发生的另一个办法是设置较大的内存工作区。

第 6 章　进程与存储管理示例

6.1　简述 Linux 系统进程的概念。

答：在 Linux 系统中，进程被赋予了下述特定的含义和特性：

(1) 一个进程是对一个程序的执行。

(2) 一个进程的存在意味着存在一个 task_struct 结构，它包含着相应的进程控制信息。

(3) 一个进程可以生成或消灭其子进程。

(4) 一个进程是获得和释放各种系统资源的基本单位。

6.2　Linux 进程上下文由哪几部分组成？为什么说核心程序不是进程上下文的一部分？进程页表也在核心区，它们也不是进程上下文的一部分吗？

答：进程上下文由 task_struct 结构、用户栈和核心栈的内容、用户地址空间的正文段、数据段、硬件寄存器的内容以及页表等组成。

核心页表被所有进程共享，所以不是进程上下文的一部分。

而进程页表是进程上下文的一部分。

6.3　假定在用户态下执行的某个进程用完了它的时间片，由于时钟中断的原因，核心调度一个新进程去执行。请形式化地描述出新、旧进程的上下文切换过程。

答：见图 6.1。

6.4　Linux 的调度策略是什么？调度时应该封锁中断吗？如果不封锁，会发生什么问题？

答：Linux 使用 3 种调度策略：动态优先数调度 SCHED_OTHER、先来先服务调度 SCHED_FIFO 和轮转法调度 SCHED_RR。其中动态优先数调度策略用于普通进程，后两种调度策略用于实时进程。

在调度时应封锁中断，否则在调度过程中由于中断会使进程上下文的切换出现错误。

6.5　试述进程 0 的作用。

答：进程 0 的作用有：创建用户进程(init 进程)，进行进程的调度和交换。

6.6　Linux 在哪几种情况下发生调度？

答：Linux 中发生进程调度的时机实质上只有两个：一个是进程自动放弃处理机时主动转入调度过程，另一个则是在由核心态转入用户态时，系统设置了高优先级就绪进程的强迫调度标识 need_resched 时发生调度。

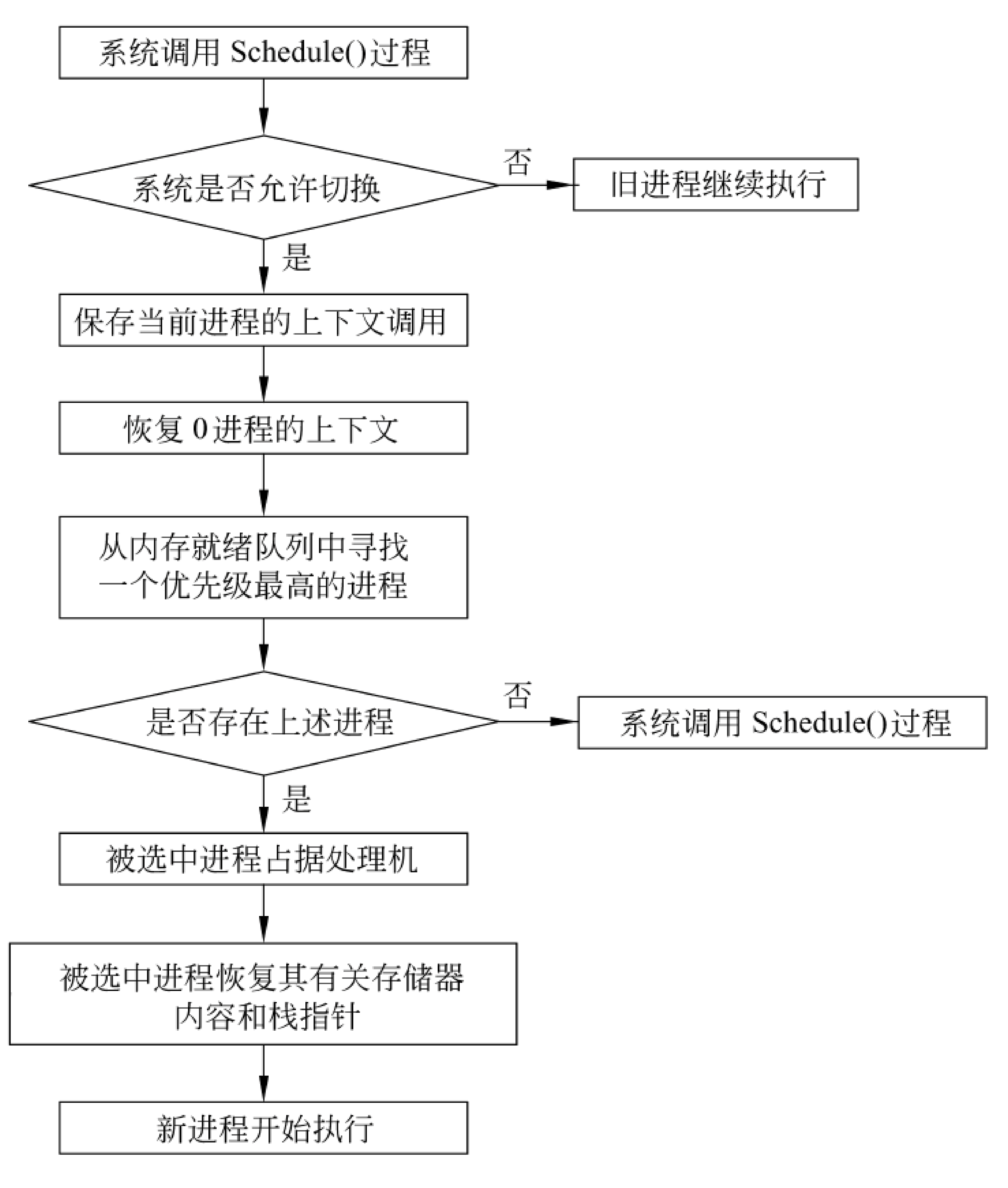


图 6.1　进程间的上下文切换过程

6.7　编写一个程序，利用 fork 调用创建一个子进程，并让该子进程执行一个可执行文件。

答：程序代码如下：

```
#include <stdio.h>
main()
{
    char * command;
    char * prompt="$ ";

    while(printf("% s",prompt),gets(command)!=NULL)
    {
        if(fork()==0)
            execlp(command,command,(char * )0);
        else
            wait(0);
    }
}
```

6.8　什么是软中断？

答：软中断是对硬件中断的一种模拟，发送软中断就是向接收进程的 task_struct 结构

中的相应项发送一个特定意义的信号。接收进程在收到软中断信号后，将按照事先的规定去执行一个软中断处理程序。但是，软中断处理程序不像硬中断处理程序那样，收到中断信号后立即被启动。它必须等到接收进程执行时才能生效。另外，一个进程自己也可以向自己发送软中断信号，以便在某些意外的情况下，进程能转入规定好的处理程序。

6.9　进程在什么时候处理它接收到的软中断信号？进程接收到软中断信号后放在什么地方？

答：进程在再次被调度执行时先检查是否收到软中断，若进程接收到了软中断信号则优先处理软中断。进程把接收到软中断信号存放在 task_struct 结构的相应项中。

6.10　Shell 符号>>将输出追加到一个指定的文件中。如果指定文件不存在，则该命令创建该文件并将输出写入其中；否则，它打开该文件并在该文件中数据尾部接着写入。编写实现>>的 C 语言代码。

答：

```
#include<stdio.h>
main()
{
    FILE * fp;
    char * filename;
    char * string;
    if(fp=fopen(filename,"a")==0) fp=fopen(filename,"w");
    fputs(string,fp);
    fclose(fp);
}
```

6.11　编写一个程序，比较使用共享存储区和消息机制进行数据传输的速度。

答：略(参见实验 2)。

6.12　形式化地描述 Linux 中消息机制的通信原理。

答：消息机制提供 4 个系统调用：msgget、msgctl、msgsnd 和 msgrcv。系统调用 msgget 返回一个消息描述字 msgqid。msgqid 指定一个消息队列供其他 3 个系统调用使用。系统调用 msgctl 用来设置和返回与 msgqid 相关联的参数选项，以及用来删除消息描述符的相关选择项。系统调用 msgsnd 和 msgrcv 分别表示发送和接收一个消息。

使用消息机制的通信双方先要建立相同的消息队列。通信时先得到自身的 msgqid，若要发送消息则把待发消息写入消息正文部分，并指定消息类型，最后调用 msgsnd 把消息发送到消息队列上；若要接收消息则调用 msgrcv 从消息队列中取出消息。

形式化描述：略。

6.13　Linux 存储管理策略中交换和请求调页方式有何区别？

答：交换技术与请求调页策略的主要区别在于：交换技术换进换出整个进程(task_

struct 结构和共享正文段除外)，因此一个进程的大小受物理存储器的限制；而请求调页策略在内存和外存之间来回传递的是存储页而不是整个进程。从而使得进程的地址映射具有了更大的灵活性，且允许进程的大小比可用物理存储空间大得多。

6.14　在图 6.19 所示请求调页的调入处理过程中，有可能出现空闲页面链表中的页面内容不同于外存设备上的页面内容的情况，此时应取哪一个页面调入内存？为什么？

答：在请求调页的调入处理过程中，若空闲页面链表中的页面内容不同于外存设备上的页面内容，则应调入空闲页面链表中的页面内容。因为在系统运行过程中有些页被淘汰到空闲页面链表中，且其中的内容已被修改，但还没有被写入外存，当此空闲页面再分配给其他进程前要把它写入外存。若原进程再次要求调入此页，就从空闲页面链表将其取出。这样可以提高系统的效率。

6.15　简要总结 Linux 进程管理与存储管理部分的联系。

答：进程管理包括进程创建、进程调度、进程执行和进程撤销。进程创建、调度和执行需要存储管理部分为进程分配或释放内存空间，进程的撤销需要存储管理部分回收分配给撤销进程的内存空间。

存储管理系统必须决定哪个进程的哪个部分应该放在内存中，并管理那些不在内存又属于同一进程虚空间的部分。

第7章　Windows的进程与内存管理

7.1　简述Windows核心态和用户态的区别。

答：（1）用户的应用程序运行在用户态，而操作系统的内核代码和设备驱动程序则运行在核心态。处在用户态的应用程序不能对操作系统的内核数据直接访问。

（2）运行于核心态的操作系统服务可以访问所有的系统内存和所有的CPU指令，可以利用所有的计算机资源完成复杂的系统管理。Windows对用户态的应用程序所能访问的系统资源有很多限制，从而保护了核心的系统资源不受侵害。

（3）所有运行于核心态的系统服务和设备驱动程序都共享同一虚地址空间；用户态进程则拥有自己独立的虚地址空间，它不能访问系统空间中的数据，也不能直接访问其他用户进程的数据空间。

7.2　Windows操作系统由哪些系统服务？简述它们的功能。

答：Windows的核心系统服务一般包括以下几部分：

（1）Windows执行体。它是运行在核心态的系统服务，用于管理进程和线程，管理内存、安全和网络，管理设备以及进程间通信。

（2）Windows内核。它为执行体提供底层系统服务，管理线程调度、中断和意外处理、多处理器同步等。

（3）设备驱动程序。它运行在核心态，管理硬件设备和处理I/O请求。

（4）硬件抽象层。它对不同的计算机环境（主要是主板硬件）提供标准的系统封装，使得其他的系统服务在设计时实现和硬件无关。

（5）窗口和图形系统。为了实现高效的用户交互，Windows的窗口管理和图形功能也运行在核心态。

7.3　描述Windows的进程和线程的概念，解释它们的区别和联系。

答：在Windows操作系统中，线程是处理器调度的对象，而进程为线程的运行提供资源和上下文环境，保证所属的线程在进程的虚拟地址空间范围内运行。

一个Windows进程包含以下信息：唯一的进程标识、一个独立的虚拟地址空间、映射到进程虚拟地址空间的执行代码和数据、访问各种系统资源的对象句柄列表、说明与进程相关用户的安全上下文定义、安全信息和访问特权设定，并至少包含一个可执行的线程。

一个Windows线程包含以下信息：唯一的线程标识、CPU寄存器的状态数据、处理器的状态、一个在用户态执行时使用的线程栈、一个用在核心态执行时使用的线程栈，以及一个供子系统、运行库和动态链接库使用的线程本地存储空间。

线程与进程的主要区别如下：

（1）进程是资源管理的基本单位，它拥有自己的地址空间和各种资源，例如内存空间、外部设备等。线程只是处理机调度的基本单位，它只和其他线程一起共享进程资源，但自己

没有任何资源。

(2) 以进程为单位进行处理机切换和调度时,由于涉及资源转移以及现场保护等问题,将导致处理机切换时间变长,资源利用率降低。以线程为单位进行处理机切换和调度时,由于不发生资源变化,特别是地址空间的变化,处理机切换的时间较短,从而处理机效率也较高。

(3) 对用户来说,多线程可减少用户的等待时间,提高系统的响应速度。例如,当一个进程需要对两个不同的服务器进行远程过程调用时,对于无线程系统的操作系统来说需要顺序等待两个不同调用返回结果后才能继续执行,且在等待中容易发生进程调度。对于多线程系统而言,则可以在同一进程中使用不同的线程同时进行远程过程调用,从而缩短进程的等待时间。

(4) 线程和进程一样,都有自己的状态,也有相应的同步机制,不过,由于线程没有单独的数据和程序空间,因此,线程不能像进程的数据与程序那样交换到外存存储空间。从而线程没有挂起状态。

(5) 进程的调度、同步等控制大多由操作系统内核完成,而线程的控制既可以由操作系统内核进行,也可以由用户控制进行。

7.4　在 Windows 进程结构中,核心进程块和进程环境块分别起到什么作用?

答:核心进程块是执行进程块的一部分,它包含了 Windows 内核调度所属线程所需要的基本信息,如时间片、核心栈和进程基准优先级等。

进程环境块驻留在进程地址空间中,提供映像调入器、堆管理器和其他运行在用户态的动态连接库所需要的进程信息,如程序映像的基地址、用户栈信息和线程的局部存储空间。

7.5　简述 Windows 的线程结构以及它在内存中的驻留机制。

答:Windows 的线程结构包括该线程所属的进程、线程创建和结束的时间、线程运行的起始例程的地址、线程级别的安全控制和等待处理的 I/O 请求包列表等。其中一部分数据被称作核心线程块,用来存储用于处理器调度线程的相关信息,如执行时间、优先级和核心栈的地址等。还有一部分被称作线程环境块的线程数据驻留在进程地址空间中,它为调入映像和动态连接库提供上下文信息。

Windows 的线程以执行线程块的形式驻留在内存中。

7.6　通过 Windows 任务管理器观察和分析系统中的进程。

答:略。

7.7　用 C 语言编写程序利用 CreateProcess 和 CreateThread 函数创建一个 Windows 进程和两个线程。

答:程序代码如下:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
```

```
DWORD WINAPI ThreadFunc(LPVOID lpParam)
{
    printf("第%d个线程创建成功.\n结束线程请输入数字%d\n",
        *(DWORD*)lpParam, *(DWORD*)lpParam);
    return 0;
}

void main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD dwFstThreadId,dwSndThreadId, dwThrdParam;
    HANDLE hFstThread,hSndThread;

    ZeroMemory(&si, sizeof(si));
    si.cb= sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // Start the child process
    if(!CreateProcess(NULL,              // No module name (use command line)
        TEXT("notepad.exe"),             // Command line
        NULL,                            // Process handle not inheritable
        NULL,                            // Thread handle not inheritable
        FALSE,                           // Set handle inheritance to FALSE
        0,                               // No creation flags
        NULL,                            // Use parent's environment block
        NULL,                            // Use parent's starting directory
        &si,                             // Pointer to STARTUPINFO structure
        &pi)                             // Pointer to PROCESS_INFORMATION structure
        )
    {
        printf("进程创建失败,错误号(%d).\n", GetLastError());
        return;
    }
    printf("进程创建成功,请关闭记事本结束进程.\n");
    // Wait until child process exits
    WaitForSingleObject(pi.hProcess, INFINITE);

    // Close process and thread handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    // first thread
    dwThrdParam= 1;
```

```
    hFstThread= CreateThread(
        NULL,                       // default security attributes
        0,                          // use default stack size
        ThreadFunc,                 // thread function
        &dwThrdParam,               // argument to thread function
        0,                          // use default creation flags
        &dwFstThreadId);            // returns the thread identifier

    // Check the return value for success

    if (hFstThread= = NULL)
    {
        printf("第一个线程创建失败 (% d)\n", GetLastError());
    }
    else
    {
        while (getch()!= '1');
        CloseHandle(hFstThread);
    }

    // second thread
    dwThrdParam= 2;
    hSndThread= CreateThread(
        NULL,                       // default security attributes
        0,                          // use default stack size
        ThreadFunc,                 // thread function
        &dwThrdParam,               // argument to thread function
        0,                          // use default creation flags
        &dwSndThreadId);            // returns the thread identifier
    if (hSndThread= = NULL)
    {
        printf("第二个线程创建失败 (% d)\n", GetLastError());
    }
    else
    {
        while (getch()!= '2');
        CloseHandle(hSndThread);
    }
}
```

7.8 简述哪些系统服务参与了 Windows 创建进程的过程？它们分别起到什么作用？

答：创建进程的过程是由 3 个部分配合完成的：创建进程的系统服务、Windows 子系统和新的进程。最常用的进程创建函数是 CreateProcess，它通过调用相应进程管理服务找到执行文件的映像，创建进程和初始线程对象，并通过消息通知 Windows 子系统新的进程和

线程对象已被创建。Windows 子系统在安装了新的进程和线程后,通知进程管理器执行初始线程。初始线程将新的进程初始化,并开始执行设定的代码。

7.9 在 Windows 处理器调度的过程中,线程的哪些状态可以转换到就绪状态?它们在什么条件下转换到该状态?

答:在 Windows 处理器调度的过程中,可以转换到就绪状态的线程状态包括初始、预备、运行、等待和过渡。

转换条件略。

7.10 简述 32 位的 Windows 操作系统的虚拟地址空间的布局。

答:如下表所示。

地址范围	大小	功能
0x0000,0000~0x7FFF,FFFF	2GB	进程的私有地址空间(程序代码、全局变量和线程栈等)
0x8000,0000~0x9FFF,FFFF	512MB	系统内核(NTLDR, HAL)和引导驱动
0xA000,0000~0xA2FF,FFFF	48MB	系统映射视图或会话空间
0xA300,0000~0xA3FF,FFFF	16MB	终端服务的系统映射视图
0xA400,0000~0xBFFF,FFFF	448MB	附加系统页表入口或附加系统高速缓存
0xC000,0000~0xC03F,FFFF	4MB	进程页表
0xC040,0000~0xC07F,FFFF	4MB	工作集链表
0xC080,0000~0xC0BF,FFFF	4MB	未使用
0xC0C0,0000~0xC0FF,FFFF	4MB	系统工作集链表
0xC100,0000~0xE0FF,FFFF	512MB	系统高速缓存
0xE100,0000~0xEAFF,FFFF	160MB	分页缓冲池
0xEB00,0000~0xFFBD,FFFF	331MB	系统页表入口和非分页缓冲池
0xFFBE,0000~0xFFFF,FFFF	4MB	故障处理和硬件抽象层(HAL)结构

7.11 简述虚拟地址到物理内存地址的转换过程,其中页表起到什么作用?

答:虚拟地址到物理内存地址的转换过程如下。

第一步:每一个进程都对应一个页目录,当操作系统开始执行某一进程时,系统会设置当前进程所对应的页目录。

第二步:一个进程可以有多个页表,通过页目录索引,内存管理器可以定位相应的虚拟地址所对应的页表。

第三步:通过页表和页表索引,内存管理器可以定位虚拟地址对应的物理页框号。

第四步:当定位了物理页框号后,通过字节索引可以正确地判断虚拟地址对应的物理地址。

页表的作用是:首先通过页表查到该虚拟地址的页表入口,再通过页表入口查到该地

址对应的物理内存地址。

7.12　简述工作集在 Windows 内存管理中的作用和工作过程。

答：工作集是驻留在物理内存中的虚拟页的集合，工作集分为进程工作集和系统工作集。进程工作集用来描述一个进程中的所有线程引用的驻留在内存中的页面，系统工作集表示系统空间中可被分页的系统代码和数据驻留在物理内存中的部分。

工作过程略。

第 8 章　文件系统

8.1　什么是文件和文件系统？文件系统有哪些功能？

答：在计算机系统中，文件被解释为一组赋名的相关字符流的集合或者是相关记录的集合。

文件系统是操作系统中与管理文件有关的软件和数据。

文件系统的功能是为用户建立文件，撤销、读写、修改和复制文件，以及完成对文件的按名存取和进行存取控制。

8.2　文件系统一般按什么分类？可以分为哪几类？

答：文件系统一般按性质、用途、组织形式、文件中的信息流向或文件的保护级别等分类。按文件的性质与用途可以分为系统文件、库文件和用户文件。按文件的组织形式可以分为普通文件、目录文件和特殊文件。按文件中的信息流向可以分为输入文件、输出文件和输入输出文件。按文件的保护级别可以分为只读文件、读写文件、可执行文件和不保护文件。

8.3　什么是文件的逻辑结构？什么是记录？

答：文件的逻辑结构就是用户可见的结构，可分为字符流式的无结构文件和记录式的有结构文件两大类。

记录是一个具有特定意义的信息单位，它由该记录在文件中的逻辑地址（相对位置）与记录名所对应的一组关键字、属性及其属性值所组成。

8.4　设文件的结构为多重结构和转置结构的组合，且定义函数 decode(K，x)和 retrive(K，x)如下：

函数 decode(K，x)为关键字 K 的搜索函数。其中 K 为待搜索关键字名，x 为顺序指针。当被搜索文件为多重结构时，指向关键字 K 的指针只有一个，即 $e(K)$，此时 $x=$nil，且 decode(K，x)返回指针 $e(K)$。当被搜索文件为转置结构时，一般指向关键字 K 的指针有 n 个，即 $e_1(K)$，…，$e_n(K)$，n 为包含关键字 K 的记录个数。此时，若 $x=$nil，decode(K，x)返回 $e_1(K)$；若 $x=e_n(K)$，则 decode(K，x)返回 nil；否则，若 $x=e_i(K)$，$1\leqslant i<n$，则 decode(K，x)返回 $e_{i+1}(K)$。

函数 retrive(K，x)是记录搜索函数。其中 K 为指向关键字的指针，x 为记录顺序指针。如果 $x=$nil 则 retrive(K，x)返回被搜索记录队列的第一个记录的逻辑地址；若 x 等于该记录队列的最后一个记录的话，则 retrive(K，x)返回 nil；否则，retrive(K，x)返回 x 的下一个记录的逻辑地址。

试问：

(1) 如果 decode(K，x)采用线性搜索法，怎样描述 decode(K，x)？

(2) 如果 retrive(K, x) 采用二分搜索法，应怎样排列记录队列？怎样描述函数 retrive(K, x)？

(3) 设函数 search(K)给出含有关键字 K 的所有记录的逻辑地址，试描述 search(K)。

答：(1) decode(K,x)的算法描述如下：

```
decode(K,x)
{
    int i
    if (文件是多重结构)return e(K)
    if (x==nil) return e1(K)
    if (x==en(K)) return nil
    for (i=1;i<n;i++)
        if (x==ei(K)) return ei+1(K)
}
```

(2) 应该对记录进行排序。retrive(K,x)算法描述如下：

```
retrive(K,x)
{
    if (x==nil) return 第一个记录的逻辑地址
    if (x==最后一个记录) return nil
    if (x==中间记录) return 中间记录后的那个记录
    if (x<中间记录) 对第一个记录和中间记录之间的记录递归调用 retrive(K,x)
    对中间记录和最后一个记录之间的记录递归调用 retrive(K,x)
}
```

(3) search(K)算法描述如下：

```
search(K)
{
    调用 decode(K,x)获得指向关键字 K的指针 e(K)
    调用 retrive(K,x)返回所有记录
}
```

8.5 设散列函数 $h(n)=(676\times l_1+26\times l_2+l_3)(\bmod\ t)$，$t=11$，$l_i$ 为关键字 n 的第 i 个英文字母序号。关键字表长为 11，关键字名为英文字母。先按线性散列法，再按平方散列法计算将任意 7 个关键字放入链表中所用的计算次数。这里令 $a=2$，$c=1$。

答：设 7 个关键字为 zhl、ouy、lwj、yks、lxz、suy 和 hls。$h(n)=(676\times l_1+26\times l_2+l_3)$ mod 11。

(1) 线性散列

$h_i(k)=(h_1(k)+2_i)$ mod 11

$h(\text{zhl})=(676\times 19+26\times 8+12)$ mod $11=9$

$h(\text{ouy})=(676\times 15+26\times 21+25)$ mod $11=8$

$h(\text{lwj})=(676\times 12+26\times 23+10)$ mod $11=8$

$h_2=(8+2\times 2)$ mod $11=1$

$h(\text{yks})=(676\times25+26\times11+19) \bmod 11=1$

$h_2=(1+2\times2) \bmod 11=5$

$h(\text{lxz})=(676\times12+26\times24+26) \bmod 11=6$

$h(\text{suy})=(676\times19+26\times21+25) \bmod 11=6$

$h_2=(6+2\times2) \bmod 11=10$

$h(\text{hls})=(676\times8+26\times12+19) \bmod 11=8$

$h_2=(8+2\times2) \bmod 11=1$

$h_3=(8+2\times3) \bmod 11=3$

计算了 12 次。

(2) 平方散列

$h_i(k)=(h_1(k)+i^2) \bmod 11$

$h(\text{zhl})=9$

$h(\text{ouy})=8$

$h(\text{lwj})=8$

$h_2=(8+2^2) \bmod 11=1$

$h(\text{yks})=1$

$h_2=(1+2^2) \bmod 11=5$

$h(\text{lxz})=6$

$h(\text{suy})=6$

$h_2=(6+2^2) \bmod 11=10$

$h(\text{hls})=8$

$h_2=(8+2^2) \bmod 11=1$

$h_3=(8+3^2) \bmod 11=6$

$h_4=(8+4^2) \bmod 11=2$

计算了 13 次。

8.6 设关键字表按英文字母顺序排列，且两关键字之间的距离为 d，写出 $n=3$ 的三分搜索算法。

答：设第一个关键字为 K_0，最后一个关键字为 K_n，要搜索的关键字为 K，则描述算法如下：

```
key findkey(K0,Kn,K)
{
    if (K==K0) return K0;
    if (K==Kn) return Kn;
    K1=K0+(Kn-K0)/3;
    K2=K0+2*(Kn-K0)/3;
    if (K==K1) return K1;
    if (K==K2) return K2;
    if(K0<K<K1) return findkey(K0,K1,K);
    if(K1<K<K2) return findkey(K1,K2,K);
```

```
    return findkey(K2,Kn,K);
}
```

8.7　文件的物理结构有哪几种？为什么说串联文件结构不适于随机存取？

答：文件的物理结构是指文件在存储设备上的存放方法。常用的文件物理结构有连续文件、串联文件和索引文件 3 种。

串联文件结构用非连续的物理块来存放文件信息。这些非连续的物理块之间没有顺序关系，链接成一个串联队列。搜索时只能按队列中的串联指针顺序搜索，存取方法应该是顺序存取的。否则，为了读取某个信息块而造成的磁头大幅度移动将花去较多的时间。因此，串联文件结构不适于随机存取。

8.8　设索引表长度为 13，其中 0～9 项为直接寻址方式，后 3 项为间接寻址方式，试描述出给定文件长度 *n*（块数）后的索引方式寻址算法。

答：设每个物理块的大小为 k 字节，每个物理块可以放 m 个物理块号。输入为文件的偏移地址 o_addr，输出为物理地址 p_addr，－1 表示寻址失败。描述算法如下：

```
addr find(o_addr)
{
    if((o_addr<0)||(o_addr>n*K-1))return(-1)
    else if(o_addr<10*K) 直接寻址,获得 p_addr
        else 间接寻址,获得 p_addr
        return(p_addr)
}
```

8.9　常用的文件存储设备的管理方法有哪些？试述主要优缺点。

答：文件存储设备的管理实质上是一个空闲块的组织和管理问题。有 3 种不同的空闲块管理方法。即空闲文件目录、空闲块链和位示图。

空闲文件目录管理方法就是把文件存储设备中的空闲块的块号统一放在一个称为空闲文件目录的物理块中，其中空闲文件目录的每个表项对应一个由多个空闲块构成的空闲区。该方法实现简单，适于连续文件结构的文件存储区的分配与回收。但是由于回收时不进行合并，所以使用该方法容易产生大量的小块空闲区。

空闲块链法把文件存储设备上的所有空闲块链接在一起，从链头分配空闲块，把回收的空闲块插入到链尾。该方法不占用额外的空间，但实现复杂。

位示图法是从内存中划出若干字节，每一位对应一个物理块的使用情况。如果该位为 0 则表示对应的是空闲块，为 1 则表示对应的物理块已分配出去。位示图法在查找空闲块时无须启动外设，但要占用内存空间。

8.10　试述成组链法的基本原理，并描述成组链法的分配与释放过程。

答：成组链法首先把文件存储设备中的所有空闲块按 50 块一组进行分组。组的划分是从后往前进行的。其中，每组的第一块用来存放前一组中各块的块号和总块数。第一组为 49 块。最后一组的物理块号与总块数只能放在管理文件存储设备用的文件资源表中。

分配和释放过程如下。

首先，系统在初始化时把文件资源表复制到内存，从而把文件资源表中放有最后一组空闲块块号与总块数的堆栈载入内存，并使得空闲块的分配与释放可以在内存中进行。用于空闲块分配与回收的堆栈有栈指针 Ptr，且 Ptr 的初值等于该组空闲块的总块数。当申请者申请 n 块空闲块时，按照后进先出的原则，分配程序在取走 Ptr 所指的块号之后，再做 Ptr＝Ptr－1 的操作。这个过程一直持续到所要求的 n 块空间都分配完毕或堆栈中只剩下最后一个空闲块的块号时。当堆栈中只剩下最后一个空闲块号时，系统启动设备管理程序，将该块中存放的下一组的块号与总块数读入内存之后再把该块分配给申请者。然后，系统重新设置 Ptr 指针，并继续为申请者分配空间。

文件存储设备的最后一个空闲块中设置有尾标识，以指示空闲块分配完毕。

如果用户进程不再使用有关文件并删除这些文件时，回收程序回收这些文件占用的物理块。成组链法的回收过程仍然利用文件管理堆栈进行。在回收时，回收程序先做 Ptr＝Ptr＋1 操作，然后把回收的物理块号放入当前 Ptr 指针所指的位置。如果 Ptr 等于 50，则表示该组已经全部回收。此时，如果还有物理块需要回收的话，那么回收该块并启动 I/O 设备管理程序，把回收的 50 个块号与块数写入新回收的块中。然后将 Ptr 置 1 另起一个新组。

对空闲块的分配和释放必须互斥进行。

8.11　什么是文件目录？文件目录中包含哪些信息？

答：一个文件的文件名和对该文件实施控制管理的说明信息称为该文件的说明信息，又称为该文件的目录。

文件目录中包含文件名、与文件名相对应的文件内部标识以及文件信息在文件存储设备上第一个物理块的地址等信息。另外还可能包含关于文件逻辑结构、物理结构、存取控制和管理等信息。

8.12　二级目录和多级目录的好处是什么？符号文件目录表和基本文件目录表是二级目录吗？

答：二级目录和多级目录的好处是可以减少文件命名冲突和提高对目录表的搜索速度。

符号文件目录表和基本文件目录表是实现文件共享的一种方法，并不是二级目录。

8.13　文件存取控制方式有哪几种？试比较它们各自的优缺点。

答：文件存取控制方式一般有存取控制矩阵、存取控制表、口令和密码 4 种方式。

存取控制矩阵方式以一个二维矩阵来进行存取控制。矩阵的一维是所有的用户，另一维是所有的文件。对应的矩阵元素则是用户对文件的存取控制权。存取控制矩阵的方法在概念上比较简单，但是当用户和文件较多时，存取控制矩阵将变得非常庞大，从而时间和空间的开销都很大。

存取控制表方式以文件为单位，把用户按某种关系划分为若干组，同时规定每组的存取限制。这样，所有用户组对文件权限的集合就形成了该文件的存取控制表。存取控制表方

式占用空间较小，搜索效率也较高，但要对用户分组，引入了额外的开销。

口令方式有两种。一种是当用户进入系统，为建立终端进程时获得系统使用权的口令。另一种口令方式是，每个用户在创建文件时，为每一个创建的文件设置一个口令，且将其置于文件说明中。当任一用户想使用该文件时，都必须首先提供口令。口令方式比较简单，占用的内存单元以及验证口令所费时间都非常少。不过，相对来说，口令方式保密性能较差。

密码方式在用户创建源文件并写入存储设备时对文件进行编码加密，在读出文件时对其进行译码解密。加密方式具有保密性强的优点。但是，由于加密和解密工作要耗费大量的处理时间，因此，加密技术是以牺牲系统开销为代价的。

8.14　设文件 SQRT 由连续结构的定长记录组成，每个记录的长度为 500B，每个物理块长 1000B，且物理结构是连续结构并采用直接存取方式；试按照图 7.23 所示的文件系统模型，写出系统调用 Read(SQRT，5，15000)的各层执行结果。其中，SQRT 为文件名，5 为记录号，15000 为内存地址。

答：第 1 层用户接口层把系统调用转化成内部调用格式。

第 2 层符号文件系统层把第 1 层提供的用户文件名转化成系统内部的唯一标识符 fd。

第 3 层基本文件系统层根据参数 fd 找到文件的说明信息。

第 4 层存取控制验证层根据存取控制信息和用户访问要求，检验文件访问的合法性。

第 5 层逻辑文件系统层根据文件逻辑结构找到第 5 个记录对应的逻辑地址 2000，并将其转换为相对块号 2。

第 6 层物理文件系统层根据文件的物理结构把相对块号 2 转换成物理地址，如 1000000。

第 7 层文件存储设备分配模块和设备策略模块把物理块号转换成具体磁盘的柱面号、磁道号和扇区号，然后准备启动输入设备命令。

第 8 层启动输入输出层由设备处理程序执行读操作，把第 5 个记录读到内存地址 15000 处。

第9章　设备管理

9.1　设备管理的目标和功能是什么?

答:设备管理的目标是:选择和分配输入输出设备以便进行数据传输操作;控制输入输出设备和CPU(或内存)之间交换数据;为用户提供一个友好的透明接口;提高设备和设备之间、CPU和设备之间以及进程和进程之间的并行操作,以使操作系统获得最佳效率。

设备管理的功能是:提供和进程管理系统的接口;进行设备分配;实现设备和设备、设备和CPU等之间的并行操作;进行缓冲区管理。

9.2　数据传送控制方式有哪几种?试比较它们各自的优缺点。

答:数据传送控制方式有程序直接控制方式、中断控制方式、DMA方式和通道方式4种。

程序直接控制方式就是由用户进程直接控制内存或CPU和外围设备之间的数据传送。它的优点是控制简单,也不需要多少硬件支持。它的缺点是:CPU和外围设备只能串行工作;设备之间只能串行工作;无法发现和处理由于设备或其他硬件所产生的错误。

中断控制方式是利用向CPU发送中断的方式控制外围设备和CPU之间的数据传送。它的优点是大大提高了CPU的利用率,且能支持多道程序和设备的并行操作。它的缺点是:由于数据缓冲寄存器比较小,如果中断次数较多,仍然占用了大量CPU时间;在外围设备较多时,由于中断次数的急剧增加,可能造成CPU无法响应中断而出现中断丢失的现象;如果外围设备速度比较快,可能会出现CPU来不及从数据缓冲寄存器中取走数据而丢失数据的情况。

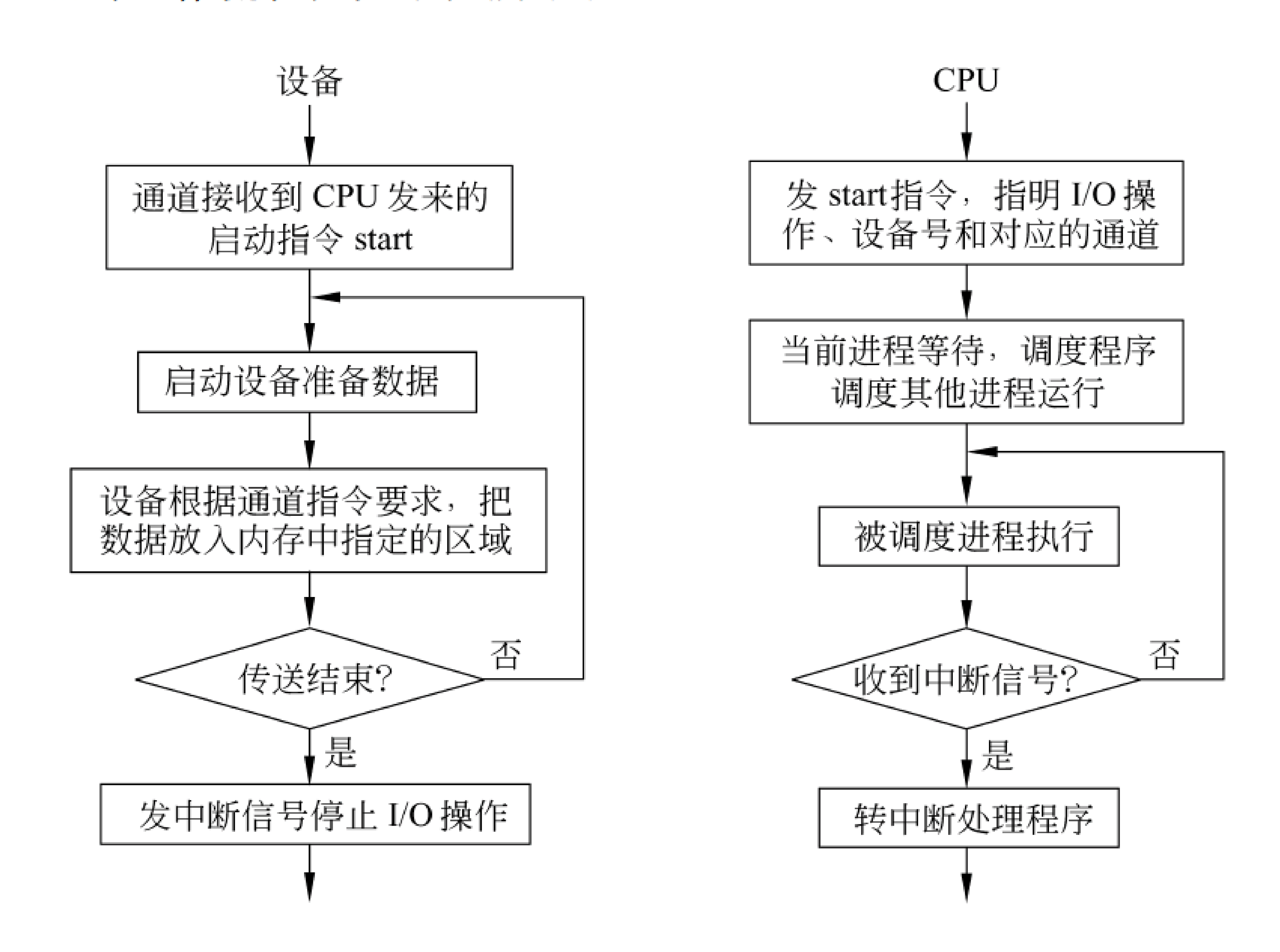
DMA方式是在外围设备和内存之间开辟直接的数据交换通路进行数据传送。它的优点是除了在数据块传送开始时需要CPU的启动指令,在整个数据块传送结束时需要发中断通知CPU进行中断处理之外,不需要CPU的频繁干涉。它的缺点是:在外围设备越来越多的情况下,多个DMA控制器的同时使用会引起内存地址的冲突,并使得控制过程进一步复杂化。

通道方式是使用通道来控制内存或CPU和外围设备之间的数据传送。通道是一个独立于CPU的专管输入输出控制的机构,它控制设备与内存直接进行数据交换。它有自己的通道指令,这些指令受CPU启动,并在操作结束时向CPU发中断信号。该方式的优点是进一步减轻了CPU的工作负担,增加了计算机系统的并行工作程度。缺点是增加了额外的硬件,造价昂贵。

9.3　什么是通道?试画出通道控制方式时的CPU、通道和设备的工作流程图。

答:通道是一个独立于CPU的专管输入输出控制的机构,它控制设备与内存直接进行数据交换。它有自己的通道指令,这些指令受CPU启动,并在操作结束时向CPU发中断信号。

设备和通道的工作流程图如下图所示。



9.4　什么叫中断？什么叫中断处理？什么叫中断响应？

答：中断是指计算机在执行期间，系统内发生任何非寻常的或非预期的急需处理事件，使得 CPU 暂时中断当前正在执行的程序而转去执行相应的事件处理程序，待处理完毕后又返回原来被中断处继续执行的过程。

CPU 转去执行相应的事件处理程序的过程称为中断处理。

CPU 收到中断请求后转到相应的事件处理程序称为中断响应。

9.5　什么叫关中断？什么叫开中断？什么叫中断屏蔽？

答：把 CPU 内部的处理机状态字（PSW）的中断允许位清除从而不允许 CPU 响应中断叫做关中断。

设置 CPU 内部的处理机状态字（PSW）的中断允许位从而允许 CPU 响应中断叫做开中断。

中断屏蔽是指在中断请求产生之后，系统用软件方式有选择地封锁部分中断而允许其余部分的中断仍能得到响应。

9.6　什么是陷阱？什么是软中断？试述中断、陷阱和软中断之间的异同。

答：陷阱指处理机和内存内部产生的中断，它包括程序运算引起的各种错误，如地址非法、校验错和页面失效。存取访问控制错、从用户态到核心态的切换等都是陷阱的例子。

软中断是通信进程之间用来模拟硬中断的一种信号通信方式。

9.7　描述中断控制方式时的 CPU 动作过程。

答：（1）首先，CPU 检查响应中断的条件是否满足。如果中断响应条件不满足，则中断处理无法进行。

(2) 如果 CPU 响应中断,则 CPU 关中断。
(3) 保存被中断进程现场。
(4) 分析中断原因,调用中断处理子程序。
(5) 执行中断处理子程序。
(6) 退出中断,恢复被中断进程的现场或调度新进程占据处理机。
(7) 开中断,CPU 继续执行。

9.8 什么是缓冲? 为什么要引入缓冲?

答: 缓冲是使用专用硬件缓冲器或在内存中划出一个区域用来暂时存放输入输出数据的器件。

引入缓冲是为了匹配外设和 CPU 之间的处理速度,减少中断次数和 CPU 的中断处理时间,同时解决 DMA 或通道方式时的数据传输瓶颈问题。

9.9 设在对缓冲队列 em、in 和 out 进行管理时,采用最近最少使用算法存取缓冲区,即在把一个缓冲区分配给进程之后,只要不是所有其他的缓冲区都在更近的时间内被使用过,则该缓冲区不再分配出去。试描述过程 take_buf(type, number)和 add_buf(type, number)。

答: 对每个缓冲区设置一个时间标志位,其取值为该缓冲区上次放入队列时的系统时间。

```
take_buf(type,number)
{
    取出时间标志位最小的缓冲区
}
add_buf(type,number)
{
    把缓冲区放入队列,并获取当前系统时间赋予其时间标志位
}
```

9.10 试述对缓冲队列 em、in 和 out 采用最近最少使用算法对改善 I/O 操作性能有什么好处。

答: 采用最近最少使用算法可以保留那些在最近一段时间内使用次数较多的缓冲区,而这些缓冲区继续被使用的可能性比较大,从而可以减少缓冲区分配和回收的次数,避免了频繁的分配和回收操作,所以可以改善 I/O 操作性能。

9.11 用于设备分配的数据结构有哪些? 它们之间的关系是什么?

答: 用于设备分配的数据结构有设备控制表(DCT)、系统设备表(SDT)、控制器表(COCT)和通道控制表(CHCT)。

SDT 整个系统一张,每个设备有一张 DCT,每个控制器有一张 COCT,每个通道有一张 CHCT。SDT 中有一个 DCT 指针,DCT 中有一个 COCT 指针,COCT 中有一个 CHCT

指针。

9.12　设计一个设备分配的安全检查程序，以保证把某台设备分配给某进程时不会出现死锁。

答：参见教材第3章3.8节。

9.13　什么是I/O控制？它的主要任务是什么？

答：I/O控制是指从用户进程的输入输出请求开始，给用户进程分配设备和启动有关设备进行I/O操作，并在I/O操作完成之后响应中断，直至善后处理为止的整个系统控制过程。

9.14　I/O控制可用哪几种方式实现？各有什么优缺点？

答：I/O控制过程可用3种方式实现：作为请求I/O操作的进程实现；作为当前进程的一部分实现；由专门的系统进程——I/O进程完成。

采用第一种方式，请求对应I/O操作的进程能很快占据处理机，但要求系统和I/O操作的进程应具有良好的实时性。第二种方式不要求系统具有高的实时性，但I/O控制过程要由当前进程负责。第三种方式增加了一个额外的进程开销，但用户不用关心I/O控制过程。

9.15　设备驱动程序是什么？为什么要有设备驱动程序？用户进程怎样使用设备驱动程序？

答：设备驱动程序是驱动外部物理设备和相应DMA控制器或I/O控制器等器件，使之可以直接和内存进行I/O操作的子程序的集合。它们负责设置相应设备有关寄存器的值，启动设备进行I/O操作，指定操作的类型和数据流向等。

设备驱动程序屏蔽了直接对硬件操作的细节，为编程者提供操纵设备的友好接口。

用户进程通过调用设备驱动程序提供的接口来使用设备驱动程序。

第 10 章　Linux 文件系统

10.1　Linux 文件系统的特点是什么？简单描述一个典型的 Linux 系统的目录结构。

答：Linux 文件系统有以下特点：

（1）具有树形结构。

（2）文件是无结构的字符流式文件。

（3）文件可以动态地增长或减少。

（4）文件数据可由文件拥有者设置相应的访问权限而受到保护。

（5）外部设备，例如磁盘设备、键盘、鼠标和串口等都被看作文件。从而，设备可通过文件系统隐蔽掉设备特性。

典型的 Linux 系统目录结构如下：

/	根目录
/bin	存放常用的用户命令
/boot	存放内核及系统启动所需的文件
/dev	存放设备文件
/etc	存放配置文件
/home	用户文件的主目录
/lib	存放运行库
/media	其他文件系统的挂载点
/mnt	与/media 目录相同
/proc	proc 文件系统的挂载点，用于存放进程和系统信息
/root	超级用户（root）的主目录
/sbin	存放系统管理程序
/sys	用于存放与设备相关的系统信息
/tmp	存放临时文件
/usr	存放应用软件包的主目录
/usr/X11R6	存放 X Window 程序
/usr/bin	存放应用程序
/usr/doc	存放应用程序文档
/usr/etc	存放配置文件
/usr/games	存放游戏
/usr/include	存放 C 语言开发工具的头文件
/usr/lib	存放运行库
/usr/local	存放本地增加的应用程序
/usr/man	存放用户帮助文件
/usr/sbin	存放系统管理程序

/usr/share	存放结构独立的数据
/usr/src	存放程序源代码
/var	存放系统产生的文件,如日志等

10.2 Linux 的文件类型有几种?分别是什么?

答: Linux 文件可分为 6 种类型,它们是普通文件、目录文件、设备文件(包括字符设备文件和块设备文件)、有名管道(FIFO)、软链接和 UNIX 域套接字。其中最常见的是普通文件、目录文件和设备文件 3 类。

10.3 什么是 VFS?它有什么作用?其通用数据模型是什么?

答: VFS 即 Virtual File System 或者 Virtual Filesystem Switch,被称为虚拟文件系统。它是 Linux 内核中的一个软件层,用于给用户空间的程序提供文件系统接口。它也提供了内核中的一个抽象功能,允许不同的文件系统共存。

VFS 隐藏了各种硬件的具体细节,为所有的文件系统操作提供了统一的接口。这样,借助 VFS,在 Linux 系统中可以使用多个不同的文件系统。不同的文件系统被挂装以后,对它们的使用与传统的单一文件系统没有区别。

VFS 通用数据模型由下列主要对象组成:

(1) 超级块(super block)。存放已挂装文件系统的有关信息。

(2) 索引节点(inode)。存放关于一个具体文件的一般信息。每个索引节点分配一个索引节点号,用来指示文件系统中的指定文件。

(3) 文件(file)。存放打开文件与进程之间进行交互的有关信息。

(4) 目录项(dentry)。保存目录项与相应文件进行链接的信息。

10.4 VFS 包括哪些系统调用?分别简述其功能。

答: VFS 处理的一些系统调用包括:

mount(),umount()	挂装/卸载文件系统
sysfs()	获取文件系统信息
statfs(),fstatfs(),ustat()	获取文件系统统计信息
chroot()	更改根目录
chdir(),fchdir(),getcwd()	操纵当前目录
mkdir(),rmdir()	创建/删除目录
stat(),fstat(),lstat(),access()	读取文件状态
open(),close(),create(),umask()	打开/关闭文件
dup(),dup2(),fcntl()	对文件描述符进行操作
select(),poll()	异步 I/O 通告
read(),write(),readv(),writev(),sendfile()	进行文件 I/O 操作
readlink(),symlink()	对软链接进行操作
chown(),fchown(),lchown()	更改文件所有者
chmod(),fchmod(),utime()	更改文件属性

pipe()　　进行通信操作

10.5　简述文件系统的注册、挂装以及卸载过程。

答：详见教材 10.3 节。

10.6　ext2 文件系统的数据块寻址是如何实现的？

答：ext2 文件系统提供了一种可以在磁盘上建立每个文件块号与相应逻辑块号之间关系的方法。

磁盘索引节点 ext2_inode 的 i_block 字段是一个有 EXT2_N_BLOCKS 个元素的数组。EXT2_N_BLOCKS 的默认值为 15。这个数组实现了文件块到磁盘逻辑块的转换，它表示一个大型数据结构的初始化部分。数组的 15 个元素有 4 种不同的类型：

(1) 最初的 12 个元素包含的逻辑块号与文件最初的 12 个块对应，即对应的文件块号为 0～11。

(2) 下标 12 的元素包含一个块的逻辑块号，这个块表示逻辑块号的一个二级数组。这个数组的元素对应的文件块号从 12 到 $b/4+11$，这里 b 是文件系统的块大小(每个逻辑块号占 4B，因此需要除以 4)。因此，内核为了查找指向数据块的指针，必须先访问这个元素。然后，用另一个指向最终块(包含文件内容)的指针访问那个块。

(3) 下标 13 中的元素包含一个块的逻辑块号，而这个块包含逻辑块号的一个二级数组，这个二级数组的数组项依次指向三级数组，这个三级数组存放的才是文件块对应的逻辑块号，范围从 $b/4+12$ 到 $(b/4)^2+(b/4)+11$。

(4) 最后，下标 14 中的元素使用三级间接索引，第四级数组中存放的才是文件块号对应的逻辑块号，范围从 $(b/4)^2+(b/4)+12$ 到 $(b/4)^3+(b/4)^2+(b/4)+11$。

10.7　Linux 系统中的设备可分为几种？分别是什么？

答：Linux 系统中的设备可分为两种，即块设备和字符设备。

第 11 章　Windows 的设备管理和文件系统

11.1　Windows 的设备管理系统由哪几部分构成？它们之间的关系如何？

答：Windows 的设备管理系统构成部分及相互关系如下：

(1) I/O 管理器(I/O manager)

它是 Windows I/O 系统的核心，它提供支持 Windows 设备驱动程序的系统结构。它将应用的 I/O 请求传递到相应的设备驱动程序，并将处理的结果返回给应用。

(2) 即插即用管理器(PnP manager)

它和 I/O 管理器和总线管理驱动一起，侦测硬件设备的加入和移出，并分配相应的硬件资源。

(3) 电源管理器(power manager)

它和 I/O 管理器和相应的设备驱动程序一起，管理该设备的能耗状态。

(4) 设备驱动程序(device driver)

它为访问特定的设备提供一个 I/O 接口。设备驱动程序从 I/O 管理器得到处理指令，在处理完后会通知 I/O 管理器。如果需要其他的设备驱动程序协助完成指令，设备驱动程序会通过 I/O 管理器将指令转到相关的设备驱动程序。

(5) 注册表(registry)和 INF 文件

注册表记录了和系统相连的硬件设备描述以及初始化和配置信息，INF 文件是用来安装相关设备驱动的文件。

(6) 硬件抽象层(Hardware Abstraction Layer，HAL)

它为设备驱动程序提供一个和计算机主板硬件无关的抽象层，使得设备驱动程序可以通过统一的接口来访问计算机主板上的设备，而不用考虑不同计算机在硬件配置上的差别。

11.2　试述 I/O 管理器的功能，它如何将 I/O 请求传递到设备驱动程序？

答：I/O 管理器功能：它是 Windows I/O 系统的核心，它提供支持 Windows 设备驱动程序的系统结构。它将应用的 I/O 请求传递到合适的设备驱动程序，并将处理的结果返回给应用。

11.3　描述 Windows 设备驱动程序的构成以及其中各个例程的功能。

答：一个典型的 Windows 设备驱动程序常常包括下面的例程：

(1) 初始化例程。当 I/O 管理器将设备驱动程序调入操作系统时，会调用该例程。该例程通过填写相应的系统数据结构来注册该设备驱动程序的其他例程，并作必要的全局初始化。

(2) 设备加入例程。所有的即插即用设备都需要实现一个设备加入例程，当即插即用管理器侦测到一个新的硬件加入时，会调用该例程。该例程为新的设备分配一个新的设备对象。

(3) 调度例程。设备驱动程序通过调度例程来实现主要的功能。例如,设备、文件系统和网络设备实现的打开、关闭、读取、写入等功能例程。

(4) I/O 起始例程。当一个 I/O 请求处理开始时,I/O 管理器通过该例程来初始化与设备交换的数据。

(5) 中断服务例程。当一个设备发出中断请求时,系统的中断调度器将控制转到该例程。它用来实现需要实时处理的设备请求,并将剩下的操作留给中断服务延迟过程调用例程处理。

(6) 中断服务延迟过程调用例程。在中断处理服务例程返回调用程序后,延迟过程调用例程有更多的时间处理完 I/O 请求的操作,以减少对其他的系统服务的影响。

(7) 终止例程。在分层设备驱动程序中需要实现这一例程,当低层的设备驱动程序完成处理后,会调用该例程来通知上层设备驱动程序相应的处理结果。

(8) 调出例程。Windows 的设备驱动程序可以动态地调入和调出,当调出设备驱动程序时,I/O 管理器通过调用该例程来释放该驱动程序占用的系统资源,并将驱动程序移出内存。

11.4 同步 I/O 处理过程和异步 I/O 处理过程的主要区别是什么?

答:驱动程序初始化 I/O 处理后会马上返回 I/O 管理器。同步 I/O 操作会等待设备处理完数据,再返回到调用程序继续执行;异步 I/O 操作则会马上返回到调用程序,等 I/O 请求处理完后,再进行数据同步。

11.5 Windows 的磁盘管理由哪几部分构成?它们如何管理磁盘?

答:Windows 的磁盘管理由扇区、分区和卷构成。

Windows 将磁盘分为固定大小的"扇区",扇区的大小取决于不同的存储设备。相邻的扇区集合组成"分区",Windows 通过分区表来存储每个分区的开始扇区、大小和其他相关的特性。Windows 通过"卷"来抽象一个或几个分区,它是文件系统操作磁盘的逻辑的单元。

11.6 分析一个安装了 Windows 操作系统的计算机的文件目录结构。

答:略。

11.7 为什么 NTFS 用簇而不是用扇区来操作磁盘?它的优点是什么?

答:NTFS 基于簇而不是扇区来操作,是为了使文件系统独立于不同大小的物理扇区。

NTFS 可以通过设置较大的簇空间来支持大容量的磁盘,或者通过簇大小的调整来支持非标准物理扇区的磁盘。当磁盘分区很大时,还可以通过设置较大的簇空间来减少磁盘碎片和磁盘定位的时间。

11.8 描述一个 NTFS 文件的主控表文件记录的格式,并画出它的结构图。

答:主控文件表由一组文件记录构成,每个文件记录的大小固定为 1KB,每一个文件在主控文件表上都有一个文件记录与之对应。当文件的所有属性(包括数据属性)占用的空间

小于1KB的文件记录时，所有的数据可以直接存储在文件记录中。当文件的大小超过1KB时，NTFS会分配额外的盘区来存储文件的数据，并通过文件记录的数据属性指向额外分配的盘区。

结构图略。

11.9　描述一个大目录结构的主控文件表文件索引记录格式，并画出它的结构图。

答：主控文件记录将其目录中的文件名和子目录名进行排序，并保存在索引根属性中。另外，系统用一个固定为4KB大小的索引缓冲区来存储不能放在索引记录中的文件名。

结构图略。

11.10　NTFS的日志中记录了哪几种操作？它们的功能是什么？

答：日志中的日志记录通常有两种：一种是更新记录，说明了某一操作在更新了文件系统结构数据后，继续执行或取消执行该操作所在的事务的方法。另一种是断点记录，NTFS会定时在日志文件中写入这种记录，说明当系统在断点处崩溃时需要执行的恢复操作。

第 12 章　嵌入式操作系统简介

12.1　嵌入式操作系统的特点是什么？

答：嵌入式操作系统除了具有通用操作系统的基本特点之外，更有其特殊的部分，如高实时性、可裁剪性、高可靠性、接口统一、网络功能强大、体积小巧、固化代码、操作简单易学等特点，其中重要的包括高实时性、可裁剪性和高可靠性等。

12.2　嵌入式操作系统中任务的状态有哪些？其状态之间是如何转换的？

答：虽然不同的嵌入式操作系统中对状态的定义不尽相同，但都包括以下 3 种基本状态：

(1) 等待：任务在等待 I/O 完成或者等待某事件的发生。

(2) 就绪：任务已经得到需要运行的资源，并等待获得处理器资源。

(3) 执行：任务获得处理器和其他所有需要的资源，相关代码正在被运行。

状态之间的转换关系如下：对于处于就绪态的任务，获得 CPU 后，处于运行态。处于运行态的任务如果被高优先级任务所抢占，或者执行的时间片期满，任务又回到就绪态。运行的任务如果需要等待某些资源，任务被切换到等待态。对于等待态的任务，如果等待的资源或事件满足，就会转换为就绪态，等待被调度执行。

12.3　什么叫任务切换？任务切换的主要工作是什么？

答：任务切换是指 CPU 的控制权由运行任务转移到另外一个就绪任务时所发生的事件，当前运行任务转为就绪（或者挂起、删除）状态，另一个被选定的就绪任务成为当前任务。

任务切换的主要工作是：保存当前任务的运行环境，更新当前运行任务的状态，移动当前任务到相应队列，调度另一个任务，更改其状态为运行，恢复将要运行任务的上下文运行环境。

12.4　嵌入式操作系统中如何实现任务间通信？

答：嵌入式操作系统中一般采用 3 种方式来实现任务间通信：

(1) 共享内存机制，用于实现数据的简单共享。

(2) 信号量机制，用于任务间基本的互斥与同步。

(3) 消息队列与管道，用于同一 CPU 内多任务间消息传递。

12.5　嵌入式操作系统中的内存管理机制如何设置？

答：嵌入式操作系统中的内存管理机制必须满足实时性、可靠性和高效性的需求，一般采用非虚拟内存管理机制和虚拟内存管理机制。非虚拟内存管理机制通常采用动态内存管理方式，系统管理空闲内存块，采用最佳适应算法或首次适应算法分配空闲块，在进程提出申请的时候，系统将从第一块空闲块开始查找，找到满足需求的内存后，返回分配起始地址。

虚拟内存管理机制需要有 MMU 提供内存地址映射和寻址功能，一般采用三级或者两级请求分页管理方式，利用 MMU 完成从虚拟地址到物理地址之间的转换，并采用“按需调页”策略为任务分配内存空间。

12.6　嵌入式系统中的中断的分类有哪些？中断处理的基本过程如何？

答：嵌入式系统中采用中断方式通知系统外部事件的发生。在实际嵌入式系统的应用中，中断可以分为硬件中断、自陷和异常 3 种类别。硬件中断是通过中断请求线输入信号来请求处理机，可分为可屏蔽中断和不可屏蔽中断，采用异步的方式执行；自陷和异常统称为软件中断，是处理机内部识别并处理进程的中断过程，其处理程序以同步方式执行。

中断处理的基本过程如下：

（1）保存上下文，保存中断服务程序将要使用的所有寄存器的内容，以便退出中断服务程序之前进行恢复。

（2）若中断向量被多个设备共享，则轮询这些设备的中断状态寄存器，确定产生该中断信号的设备。

（3）获取中断相关的其他信息。

（4）对中断进行具体的处理。

（5）恢复保存的上下文。

（6）执行中断返回指令，使 CPU 的控制返回到被中断的程序继续执行。

12.7　嵌入式操作系统中文件、设备和设备驱动之间的关系如何？

答：嵌入式操作系统中的文件放在磁盘设备上，嵌入式系统中将文件子系统划分为 4 层，分别是 API 接口层、文件系统层、中间件层和物理驱动层。API 接口层是与用户的接口，文件系统层用于实现文件系统的初始化工作和与下层的格式化接口，中间件层主要实现对存储设备的管理，包括向上的文件系统层接口、地址转换、物理设备驱动接口等功能；驱动层主要实现对不同存储介质的基本访问驱动接口，所有的嵌入式存储设备提供的接口在中间件层时会进行封装，来实现具体设备之间的统一访问。因此，用户对文件的操作，会通过 API 接口，从中间层中获取磁盘设备的驱动接口，进行封装，然后在由驱动层的基本访问驱动接口来实现对磁盘的访问，完成信息的修改。

12.8　一般嵌入式系统的开发流程包括哪些步骤？

答：一般嵌入式系统的开发流程包括以下几个步骤：

（1）确定开发的硬件环境和开发调试环境。

（2）配置开发调试环境下的模板联编文件。

（3）创建新的任务，编写源代码程序。

（4）调试通过后，下载模型映像进行仿真和交互运行，验证执行结果。

综 合 试 题

操作系统综合练习试题 1

1. (10 分)试述分区式管理中的最先适应算法(FF)、最佳适应算法(BF)以及最坏适应算法(WF)的原理,并比较其优缺点。
2. (10 分)单项选择。

   (1) 虚存是______。

   ① 提高运算速度的设备　　② 容量扩大了的内存

   ③ 实际不存在的存储器　　④ 进程的地址空间及其内存扩大方法

   (2) 临界区是______。

   ① 一个缓冲区　　② 一段共享数据区

   ③ 一段程序　　④ 一个互斥资源

   (3) 在 UNIX 系统中,用户通过______读取磁盘文件中的数据。

   ① 作业申请表　　② 原语

   ③ 系统调用　　④ 中断

   (4) UNIX System Ⅴ 的调度原理是基于______。

   ① 时间片调度　　② 先来先调度

   ③ 时间片＋优先级　　④ 最短作业优先
3. (8 分)下列程序执行时,“parent: child exited”可能在“child leaving”前面打印吗?为什么?程序执行结果中 a 的值是多少?为什么?

```
{…
    a=55;
    pid=fork();
    if (pid==0)
    {
        sleep(5);
        a=99;
        sleep(5);
        printf("child leaving\n");
        exit(0);
    }
    else
    {
        sleep(7);
        printf("a==%d\n", a);
        wait(0);
```

```
            printf("parent: child exited\n");
        }
        …
    }
```

4. (10 分)设有进程 A、B、C,分别调用过程 get、copy 和 put 对缓冲区 S 和 T 进行操作。其中 get 负责把数据块输入缓冲区 S,copy 负责从缓冲区 S 中提取数据块并复制到缓冲区 T 中,put 负责从缓冲区 T 中取出信息打印(见下图)。描述 get、copy 及 put 的操作过程。

get → 缓冲区 S —copy→ 缓冲区 T —put→

5. (12 分)描述 UNIX System Ⅴ 中的缓冲区申请算法 getblk。说明为什么在相应的缓冲区标志了延迟写以后,要启动设备把该块内容写回磁盘,并分配另一个缓冲区给进程?

操作系统综合练习试题 1 解答

1. 最先适应法:把空闲区按首地址从低到高顺序排列,从表头开始搜索,直到找到空闲块大小不小于所需大小的空闲区。

最佳适应法:空闲区从小到大顺序排列,从表头开始搜索,直到找到空闲块大小不小于所需大小的空闲区。

最坏适应法:空闲区按从大到小的顺序排列,每次取第一个空闲区,如果其大小不小于所需大小,则分配,否则放弃。

各算法的优缺点如下表所示。

算法	速度	空间	特点
FF	最好	中等	使用低地址区
BF	较慢	差(易产生碎片)	最接近要求
WF	较慢	好	易于合并,碎片少

2. ④,③,③,③。

3. 由于 UNIX 输出时采用缓冲区缓冲数据,尽管本程序采用了同步机制,缓冲区中的数据“parent: child exited”仍可能在“child leaving”前面打印输出。

a=55,因为子进程和父进程的资源是独立的。

4. 发送进程 Pa,接收进程 Pb 和 Pc。

Pa 的私用信号量 Sem(判断缓冲区 S 是否为空),Pb 的私用信号量 Sfull(从缓冲区 S 接收数据);Pa 的私用信号量 Tem(判断缓冲区 T 是否为空),Pc 的私用信号量 Tfull(从缓冲区 T 中接收数据)。

初值:

Sem＝1，Sfull＝0，Tem＝1，Tfull＝0

描述：

```
Pa: get
    begin
        P(Sem)
        把数据块写入 S
        V(Sfull)
    end
Pb: copy
    begin
        P(Sfull)
        P(Tem)
        把数据从 S中提取到 T中
        V(Sem)
        V(Tfull)
    end
Pc: put
    begin
        P(Tfull)
        从 T中取数据打印
        V(Tem)
    end
```

5.

```
getblk
输入:逻辑设备号、逻辑块号
输出:加锁的缓冲区
begin
    while(未申请到缓冲区)
    do
        if(数据块在设备队列中,且该缓冲区被标记为忙)
        then 等待,进入相应的等待队列
        fi
        if(数据块在设备队列中,且被标记为空闲)
        then 将该块置为忙,并加锁返回
        fi
        if(数据块不在设备队列中,且空闲缓冲区队列为空)
        then 等待,进入等待队列
        fi
        if(数据块不在设备队列,且空闲缓冲区队列不为空)
        then  从空闲队列中取一缓冲区
              if 该缓冲区为延迟写
```

```
            then 启动 I/O,将该块内容写回磁盘
                 重新申请缓冲区
            else 上锁该缓冲区,并返回
            fi
        fi
    od
end
```

操作系统综合练习试题 2

1. 填空

   (1) 操作系统的主要特点是(　　)、(　　)和(　　)。

   (2) 在单 CPU 系统中,CPU 和(　　)是并行操作的。

   (3) 作业控制方式有(　　)和(　　)。

   (4) 操作系统为编程人员提供的接口是(　　),为一般用户提供的接口是(　　)。

   (5) UNIX 进程 0 的主要作用有(　　)和(　　)。

2. 名词解释

   同步;互斥;目录与索引节点;进程;线程;虚存

3. 设 UNIX SystemⅤ的调度优先数计算公式为

$$p_pri = p_cpu/2 + 45 + p_nice$$

   其中 p_cpu 为被计算进程最近一次使用 CPU 后的 CPU 使用时间表述值,其初始值为 0。进程占有 CPU 时,p_cpu 周期性地加 1(每秒增加 60)。进程的 p_cpu 每秒衰减 p_cpu/2。

   设 p_nice=20,且 UNIX SystemⅤ按 p_pri 值最小的进程优先获得 CPU 方式调度,进程 Pa、Pb、Pc 和 Pd 在初始时 p_cpu 为 0。处理机每秒发生一次调度。计算 0～5 秒之间,各进程在每秒发生调度时的 p_pri、p_cpu 以及占有 CPU 的进程。

4. 描述 UNIX SystemⅤ的索引节点释放过程 ifree,并指出该过程可能存在的问题。

5. UNIX 系统采用异步写与延迟写等方式将数据写回外存,试述异步写与延迟写的区别。

6. UNIX SystemⅤ对进程的用户区和系统区采用不同的存储管理方式,即系统区采用分区式管理,然后将对应部分全部装入内存。而用户区部分则采用请求调页管理。问:

   (1) 为什么要对系统区进行分区式管理?

   (2) 设内存区的分配释放采用位示图方式,且位示图在内存的控制数组结构为

```
mem struct
{
    int m_free;        /* 该组空闲页面数 */
    int m_ptr;         /* 位示图位置指针 */
    int m_avail;       /* 内存中的空闲页面数 */
```

```
        short m_pnum[NIOMEM];
    }
```

试描述内存页面分配过程 memall(base, size)。

操作系统综合练习试题 2 解答

1.

(1) 执行并发;资源共享;用户随机

(2) 输入输出设备

(3) 脱机控制;联机控制

(4) 系统调用;命令界面

(5) 交换;调度

2. 同步:异步环境下的一组并发进程因直接制约而相互发送消息,相互合作,相互等待,使得各进程按一定的速度向前推进的过程称为进程同步。具有同步关系的一组进程称为合作进程,合作进程间相互发送的信号称为消息或事件。

互斥:一组并发进程中的一个或多个程序段,因共享某一临界资源而导致它们必须以一个不允许交叉执行的单位执行。也就是说,不允许两个以上共享资源的并发进程同时进入临界区称为互斥。

进程互斥必须满足如下准则:

(1) 与各并发进程的执行速度无关,即各进程享有平等的、独立的竞争共有资源的权利。

(2) 不在临界区的进程不能阻止其他进程进入临界区。

(3) 当有若干进程申请进入临界区时,只能允许一个进程进入。

(4) 当一个进程申请进入临界区时,应在有限时间内进入。

目录与索引节点:文件名和相应的文件标识号称为目录。也有系统把文件说明信息称为目录。索引节点在 UNIX 系统或 Linux 系统中用来存放索引结构等文件说明信息和文件标识号。

进程:一个具有独立功能的程序对某个数据集在处理机上的执行过程和分配资源的基本单位。

线程:在进程的地址空间内,共享进程的各种资源,由寄存器和相应堆栈组成的处理机切换单位。

虚存:由指令的寻址方式所决定的进程寻址空间,以及根据该空间所形成的由内外存组成的存储介质。实现虚存必须满足以下 3 点:

(1) 有相应的地址变换机构,把虚拟(逻辑)地址变换为内存物理地址。

(2) 根据程序执行需要,自动选择指令进入内存。

(3) 有足够大的外存,存储进程的指令与数据,从而使得那些暂不访问的指令和数据都在外存。

3.

t	P_A pri	CPU 计数	P_B pri	CPU 计数	P_C pri	CPU 计数	P_D pri	CPU 计数	占有 CPU 的进程
0	65	0	65	0	65	0	65	0	P_A
1	80	60 30	65	0	65	0	65	0	P_B
2	72	30 15	80	60 30	65	0	65	0	P_C
3	68	15 7	72	30 15	80	60 30	65	0	P_D
4	66	7 3	68	15 7	72	30 15	80	60 30	P_A
5	80	63 31	66	7 3	68	15 7	72	30 15	P_B

4. 在 UNIX System Ⅴ 中，索引节点按从小到大方式排列，组成索引节点管理数组。其中铭记索引节点为索引节点数组中的最高序号索引节点。

在释放索引节点时，如果该索引节点数组有空位，则可将所释放的索引节点号置入该空位中。

如果该索引节点数组已满，且所释放的索引节点号＞铭记索引节点，则将该索引节点开锁后不置入索引节点数组。

若所释放的索引节点号＜铭记索引节点，则将铭记索引节点号与所释放的索引节点号交换，以便于再次分配。

算法描述：

```
ifree:
    begin
        空闲 i 节点数+1
        if i 节点数组空,且数组未上锁
        then i 节点号入数组
        fi
        if i 节点数组满
        then  比较铭记号与释放 i 节点的序号大小
              if 铭记号大
              then i 节点锁定位置空闲
              else 把索引节点的锁定位置空闲,并用该索引节点号代替铭记索引节点
              fi
              if 索引节点数组被锁定
              then 释放索引节点后直接返回
              fi
        fi
end
```

可能存在的问题：

由于索引节点数组被锁定时，系统正在从磁盘块中读写相应的索引节点号，因此，无法更改索引节点数组的铭记索引节点。此时，如果被释放的索引节点号小于铭记索引节点号的话，由于索引节点数组每次从铭记号开始，按从小到大方式读索引节点号入数组进行分配，当被释放的索引节点序号小于铭记索引节点号，且由于索引节点数组被锁定后无法进行铭记索引节点换号时，就造成了该索引节点无法进入索引节点数组进行再分配，从而漏掉该索引节点。

5. UNIX 系统采用异步写、延迟写和同步写 3 种方式，把数据从缓冲写回外存。延迟写是将装有待写数据的缓冲区放入空闲队列，在不启动写过程时就返回。待其他进程申请缓冲区时，如果分配到该缓冲区，则将其写回外存。异步写是指系统启动 I/O 后，不等待传输完成就立即返回。

异步写与延迟写的主要区别是：延迟写方式将待写数据在缓冲区中尽量多放一段时间，以减少 I/O 操作的次数。异步写方式则主要是为了提高进程的执行速度，减少进程的等待时间。

6. (1) 因为系统区主要装载系统程序和相关数据结构，而且这些系统程序和数据一旦装载入内存后就不再换出，这正适合分区式管理的特点。因此，UNIX 对系统分区采用分区式管理。

(2) memall (base, size)的描述如下：

```
memall(base, size)
begin
    if (m_avail 小于 size)
    then 分配失败,返回
    fi
    while (m_free 小于 size)
    {
        将数组 m_pnum 中的 m_free 个页面填入页表
        将位示图中的相应位置为 0
        m_avail ← m_avail-m_free
        size ← size-m_free
        从位示图中 m_ptr 所指位开始读出 NICMEM 个页面入 m_pnum 数组
        m_free ← NICMEM
    }
    从 m_pnum 中取出 size 个页面填入页表
    将位示图中的相应位置为 0
    m_avail ← m_avail-m_free
    size←size-m_free
    返回分配页面数
end
```

操作系统综合练习试题 3

1. 画出 Linux 的进程状态转换图。

2. 名词解释：

系统调用；进程；并发与并行；临界区；同步；死锁；虚存；动态地址重定位；文件系统；中断

3. UNIX 系统采用一般写、异步写和延迟写 3 种方式将缓冲区中内容写回磁盘。试述这 3 种方式各自的特点。

4. 当系统发生缺页时，试问所缺的页面数据可能在什么地方？画出相应的页面调入处理框图。

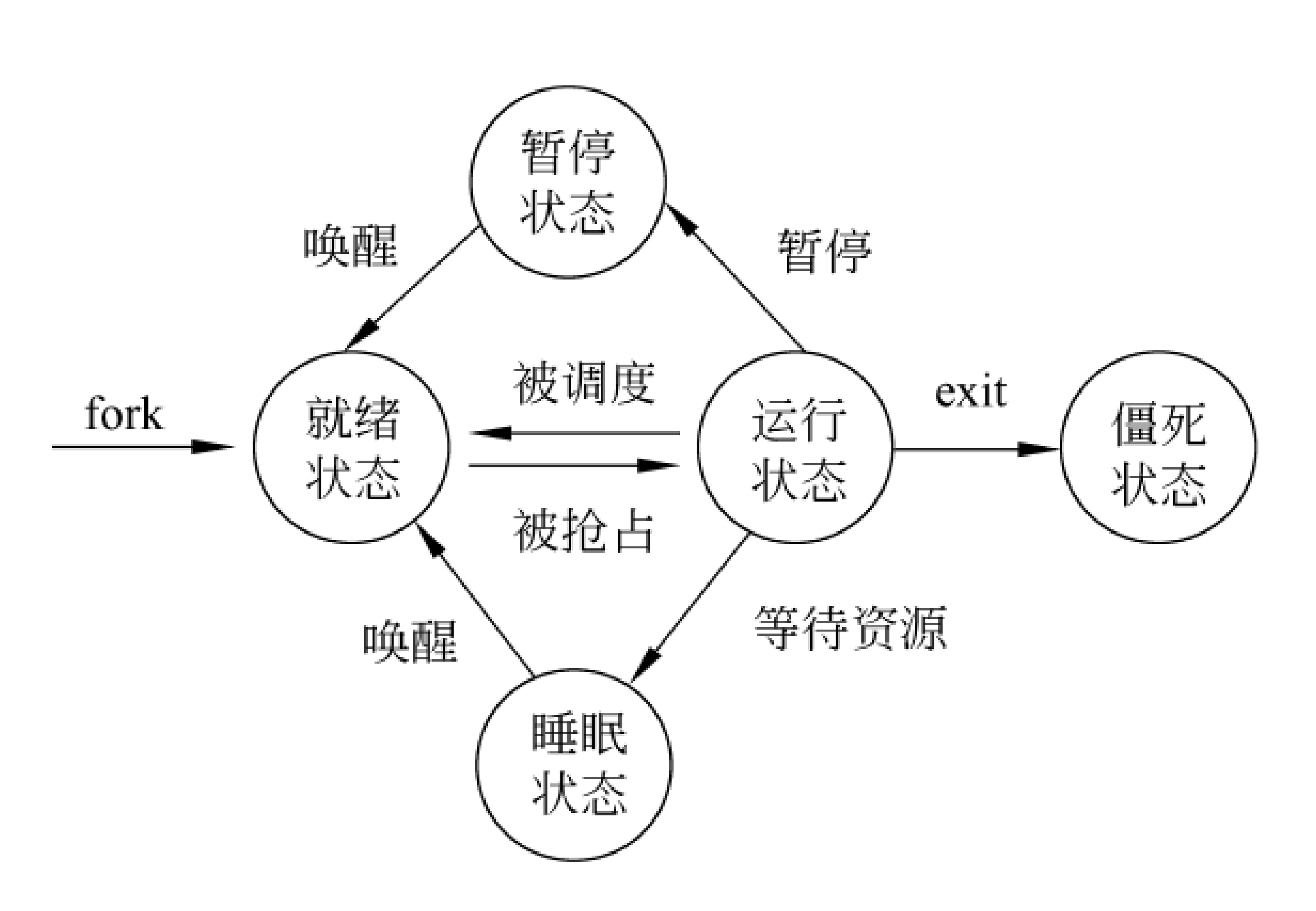
5. 试述成组链法的基本原理，并描述采用成组链法的磁盘块分配过程。

6. 父进程 PA 使用系统调用 fork()创建了一个子进程 Pa，设 Pa 中有一局部变量 V，且 V 在 Pa 创建之前已被赋值。试问：如果在 Pa 中改变 V 的值，是否会改变 Pa 中 V 的值？

7. 动态分区式管理内存时常用的内存分配与释放算法有哪 3 种？分别简述这 3 种算法的基本原理并比较其优缺点。

操作系统综合练习试题 3 解答

1.



2. 系统调用：操作系统提供给编程人员的唯一接口，其指令在核心态下执行。

进程：一个程序对数据集的执行过程。

并发与并行：一组在逻辑上互相独立的程序在执行过程中在执行时间上互相重叠的执行方式，为并发。一组程序按独立的、异常的速度执行为并行。

临界区：不允许多个并发进程交叉执行的一段程序。

同步：异步环境下的一组并发进程因直接制约而互相发送消息，进行合作，使各进程按一定的速度执行的过程。

死锁：一组并发进程因互相请求对方所拥有的资源，在无外力的条件下无法继续执行的状态。

虚存：一个进程的目标代码与数据的虚拟地址组成的空间为虚存，其大小受计算机地

址结构限制，虚存的实现具有 3 个条件：内存中只存储那些经常执行的指令；指令的虚拟地址由硬件地址自动变换机构变换成物理地址；在外存中的数据和程序根据执行需要按需调入内存。

动态地址重定位：

（1）设置 BR 和 VR。

（2）将程序段首地址装入 BR 中。

（3）将所要访问的虚拟地址送 VR。

（4）地址变换机构把 VR 和 BR 内容相加得实际地址。

文件系统：与管理文件有关的程序和数据结构。

中断：计算机执行期间，系统发生非预期急需处理事件，使得 CPU 暂时中断当前正在执行的程序，转去执行相应的事件处理程序。

3．一般写：启动设备写时进程睡眠，等到写结束后唤醒等待进程。

异步写：一次写两块，其中第一块为一般写，第二块则是不等待写结束即返回，从而提高访问外存的速度。

延迟写：等到分配装有该块数据的缓冲区时再将该块内容写入磁盘，从而增加数据在内存的驻留时间。

4．数据块在缓冲区中、交换区中或文件系统中。

算法描述如下：

```
if 所需页面在内存中
    then 转地址变换，Exit 返回
fi
if 所需页面在交换区中
    then 启动交换程序，将交换区中相应块调入内存
fi
if 所需页面在缓冲区中
    then 从缓冲区中把该块数据移入内存
fi
if 所需页面在文件系统中
    then 启动文件系统，找出该块在文件系统中的位置，把该块读入内存
fi
```

5．成组链法的基本原理如下：

（1）把文件系统中的外存存储区的空闲块按 50 块一组从后往前划分。

（2）第 n 组的总块数和块号作为数据存放在第 $n+1$ 组的最后一块中。最后一组的块数和块号放在文件资源表的堆栈中。

（3）系统初启时将文件资源表中的有关堆栈复制到内存中。

（4）文件系统空闲块的分配释放在读入内存的文件资源表管理堆栈上进行。

（5）若堆栈中存储的块号已分配完毕，只剩下最后一块时，锁住堆栈，不再进行空闲块分配。同时，启动外设将最后一块中所记载的块数和块号读入堆栈，待该块中的块数和块号全部读入堆栈后，对堆栈解锁，再继续分配。

（6）若堆栈满，且又有一个新的空闲块被释放时，锁住堆栈，且启动外设，将堆栈中所存

放的块号和块数写入新释放的块中后再打开堆栈。

```
分配过程 alloc:
输入:待分配的块数 n
输出:块号
begin
    if 堆栈 filsys 被锁定
    then 等待 filsys 被开锁
    fi
    ptr 所指块号入 val               /* ptr 为堆栈指针,val 为存放输出块号的变量 */
    堆栈指针 ptr=ptr-1
    if ptr=0                         /* 堆栈中只剩最后一个块号 */
    then  锁住 filsys
          启动外设,从 ptr 所指块号中读入块数与块号
          sleep                      /* 等待读入完成事件唤醒 */
    else 返回 val 中块号
end
```

6. 否。

7. 略。

第二部分

实 验 指 导

系统调用函数说明、参数值及定义

1．fork()

创建一个新进程。

```
int fork()
```

其中返回 int 取值意义如下：

0：创建子进程，从子进程返回的 id 值。

大于 0：从父进程返回的子进程 id 值。

－1：创建失败。

2．lockf(files，function，size)

用作锁定文件的某些段或者整个文件。

本函数使用的头文件为

```
#include <unistd.h>
```

参数定义如下：

```
int lockf(files,function,size);
int files,function;
long size;
```

其中，files 是文件描述符；function 是锁定和解锁，1 表示锁定，0 表示解锁；size 是锁定或解锁的字节数，若用 0，表示从文件的当前位置到文件尾。

3．msgget(key，flag)

获得一个消息的描述符，该描述符指定一个消息队列以便用于其他系统调用。

该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义如下：

```
int msgget(key,flag);
key_t key;
int flag;
```

语法格式：

```
msgqid=msgget(key,flag)
```

其中：
msgqid 是该系统调用返回的描述符，失败则返回－1；
flag 本身由操作允许权和控制命令值相“或”得到。例如：

```
IPC_CREATE|0400                     是否该队列应被创建
IPC_EXCL|0400                       是否该队列的创建应是互斥的
```

4．msgsnd（id，msgp，size，flag）

发送一个消息。
该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义如下：

```
int msgsnd(id,msgp,size,flag);
int id,size,flag;
struct msgbuf * msgp;
```

其中，id 是返回消息队列的描述符；msgp 是指向用户存储区的一个构造体指针；size 指示由 msgp 指向的数据结构中字符数组的长度，即消息的长度，这个数组的最大值由 MSG_MAX 系统可调用参数来确定；flag 规定当核心用尽内部缓冲空间时应执行的动作。若在标志 flag 中未设置 IPC_NOWAIT 位，则当该消息队列中的字节数超过某一最大值时，或系统范围的消息数超过某一最大值时，调用 msgsnd 进程睡眠。若是设置 IPC_NOWAIT，则在此情况下，msgsnd 立即返回。

5．msgrcv（id，msgp，size，type，flag）

接受一个消息。
该函数调用使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义如下：

```
int msgrcv(id,msgp,size,flag);
int id,size,type,flag;
struct msgbuf * msgq;
struct msgbuf{long mtype;char mtext[];};
```

语法格式如下：

```
count=msgrcv(id,msgp,size,type,flag)
```

其中，id 是消息描述符。msgp 是用来存放欲接收消息的用户数据结构的地址。size 是

msgp 中数据数组的大小。type 是用户要读的消息类型，为 0 时接收该队列的第一个消息，为正时接收类型 type 的第一个消息，为负时接收小于或等于 type 绝对值的最低类型的第一个消息。flag 规定倘若该队列无消息，核心应当做什么事。如果此时设置了 IPC_NOWAIT 标志，则立即返回；若在 flag 中设置了 MSG_NOERROR，且所接收的消息大小大于 size，核心截断所接收的消息。

count 是返回消息正文的字节数。

6．msgctl(id,cmd,buf)

查询一个消息描述符的状态，设置它的状态及删除一个消息描述符。

调用该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

参数定义如下：

```
int msgctl(id,cmd,buf);
int id,cmd;
struct msgid_ds * buf;
```

函数调用成功时返回 0，调用不成功时返回－1。其中，

id 用来识别该消息的描述符。cmd 规定命令的类型：

(1) IPC_STAT 将与 id 相关联的消息队列首标读入 buf。

(2) IPC_SET 为这个消息序列设置有效的用户和小组标识及操作允许权和字节的数量。

(3) IPC_RMID 删除 id 的消息队列。

buf 是含有控制参数或查询结果的用户数据结构的地址。

msgid_ds 结构定义如下：

```
struct msgid_ds
{
    struct ipc_perm msg_perm;          /* 许可权结构 */
    short pad1[7];                     /* 由系统使用 */
    ushort msg_qnum;                   /* 队列中的消息数 */
    ushort msg_qbytes;                 /* 队列中的最大字节数 */
    ushort msg_lspid;                  /* 最后发送消息的 PID */
    ushort msg_lrpid;                  /* 最后接收消息的 PID */
    time_t msg_stime;                  /* 最后发送消息的时间 */
    time_t msg_rtime;                  /* 最后接收消息的时间 */
    time_t msg_ctime;                  /* 最后更改时间 */
};
struct ipc_perm
{
    ushort uid;                        /* 当前用户 id */
    ushort gid;                        /* 当前进程组 id */
```

```
    ushort cuid;                    /* 创建用户 id */
    ushort cgid;                    /* 创建进程组 id */
    ushort mode;                    /* 存取许可权 */
    {short pad1;long pad2}          /* 由系统使用 */
};
```

7. shmget(key,size,flag)

获得一个共享存储区。

该函数使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

语法格式如下：

```
shmid=shmget(key,size,flag)
```

参数定义如下：

```
int shmget(key,size,flag);
key_t key;
int size,flag;
```

其中，size 是存储区的字节数，key 和 flag 与系统调用 msgget 中的参数含义相同。

flag 中包含的操作允许权和控制命令如下。

操作允许权	八进制数
用户可读	00400
用户可写	00200
小组可读	00040
小组可写	00020
其他可读	00004
其他可写	00002

控制命令	值
IPC_CREATE	0001000
IPC_EXCL	0002000

例如：

```
shmid=shmget(key,size,(IPC_CREATE|0400));
```

创建一个关键字为 key、长度为 size 的共享存储区。

8. shmat(id,addr,flag)

从逻辑上将一个共享存储区附接到进程的虚拟地址空间上。

该函数调用使用头文件如下：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义如下：

```
int * shmat(id,addr,flag);
int id,flag;
char * addr;
```

语法格式如下：

```
virtaddr=shmat(id,addr,flag)
```

其中，id 是共享存储区的标识符。addr 是用户要使用共享存储区附接的虚地址，若 addr 是 0，系统选择一个适当的地址来附接该共享区。flag 规定对此区的读写权限，以及系统是否应对用户规定的地址做舍入操作。如果 flag 中设置了 shm_rnd 即表示操作系统在必要时舍去这个地址；如果设置了 shm_rdonly，即表示只允许读操作。virtaddr 是附接的虚地址。

9．shmdt（addr）

把一个共享存储区从指定进程的虚地址空间断开。

调用该函数使用如下头文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义：

```
int shmdt(addr);
char * addr;
```

当调用成功时，返回 0 值；调用不成功，返回－1。其中，addr 是系统调用 shmdt 所返回的地址。

10．shmctl（id，cmd，buf）

对与共享存储区关联的各种参数进行操作，从而对共享存储区进行控制。

调用该函数使用如下头文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义如下：

```
int shmctl(id,cmd,buf);
```

```
int id,cmd;
struct shmid_ds * buf;
```

调用成功时返回 0,否则返回－1。其中,id 为被共享存储区的标识符。cmd 规定操作的类型,规定如下:

(1) IPC_STAT:返回包含在指定的 shmid 相关数据结构中的状态信息,并且把它放置在用户存储区中的 * buf 指针所指的数据结构中。执行此命令的进程必须有读取允许权。

(2) IPC_SET:对于指定的 shmid,为它设置有效用户和小组标识和操作存取权。

(3) IPC_RMID:删除指定的 shmid 以及与它相关的共享存储区的数据结构。

(4) SHM_LOCK:在内存中锁定指定的共享存储区,必须是超级用户才可以进行此项操作。

buf 是一个用户级数据结构地址。

shmid_ds 结构定义如下:

```
shmid_ds
{
    struct ipc_perm shm_perm;          /* 许可权结构 */
    int shm_segsz;                     /* 段大小 */
    int pad1;                          /* 由系统使用 */
    ushort shm_lpid;                   /* 最后操作的进程 id */
    ushort shm_cpid;                   /* 创建者的进程 id */
    ushort shm_nattch;                 /* 当前附接数 */
    short pad2;                        /* 由系统使用 */
    time_t shm_atime;                  /* 最后附接时间 */
    time_t shm_dtime;                  /* 最后断接时间 */
    time_t shm_ctime;                  /* 最后修改时间 */
}
```

11. signal(sig,function)

允许调用进程控制软中断信号的处理。

调用该函数使用如下头文件:

```
#include < signal.h>
```

参数定义如下:

```
signal (sig,function)
int sig;
void (* func)();
```

其中,sig 的值如下:

SIGHVP	挂起
SIGINT	键盘按 delete 键或 break 键
SIGQUIT	键盘按 quit 键
SIGILL	非法指令
SIGIOT	IOT 指令

SIGEMT	EMT 指令
SIGFPE	浮点运算溢出
SIGKILL	要求终止进程
SIGBUS	总线错
SIGSEGV	段违例
SIGSYS	系统调用参数错
SIGPIPE	向无读者管道上写
SIGALRM	闹钟
SIGTERM	软件终结
SIGUSR1	用户定义信号
SIGUSR2	第二个用户定义信号
SIGCLD	子进程死
SIGPWR	电源故障

function 的值如下：

SIG_DFL：默认操作。对除 SIGPWR 和 SIGCLD 外所有信号的默认操作是进程终结。对信号 SIGQUIT、SIGILL、SIGTRAP、SIGIOT、SIGEMT、SIGFPE、SIGBUS、SIGSEGV 和 SIGSYS，它产生一个内存映像文件。

SIG_IGN：忽视该信号的出现。

func 是在该进程中的一个函数地址，在核心返回用户态时，它以软中断信号的序号作为参数调用该函数，对除了信号 SIGILL、SIGTRAP 和 SIGPWR 以外的信号，核心自动地重新设置软中断信号处理程序的值为 SIG_DFL，一个进程不能捕获 SIGKILL 信号。

实验1　进程管理

实验目的

(1) 加深对进程概念的理解,明确进程和程序的区别。
(2) 进一步认识并发执行的实质。
(3) 分析进程争用资源的现象,学习解决进程互斥的方法。
(4) 了解 Linux 系统中进程通信的基本原理。

实验预备内容

(1) 阅读 Linux 的 sched.h 源码文件,加深对进程管理概念的理解。
(2) 阅读 Linux 的 fork.c 源码文件,分析进程的创建过程。

实验内容

1. 进程的创建

编写一段程序,使用系统调用 fork()创建两个子进程。当此程序运行时,在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符:父进程显示字符“a”;子进程分别显示字符“b”和“c”。试观察记录屏幕上的显示结果,并分析原因。

2. 进程的控制

修改已编写的程序,将每个进程输出一个字符改为每个进程输出一句话,再观察程序执行时屏幕上出现的现象,并分析原因。

如果在程序中使用系统调用 lockf()来给每一个进程加锁,可以实现进程之间的互斥,观察并分析出现的现象。

3. 进程的软中断通信

(1) 编制一段程序,使其实现进程的软中断通信。

要求:使用系统调用 fork()创建两个子进程,再用系统调用 signal()让父进程捕捉键盘上来的中断信号(即按 Del 键);当捕捉到中断信号后,父进程用系统调用 kill()向两个子进程发出信号,子进程捕捉到信号后分别输出下列信息后终止:

```
child process 1 is killed by parent!
child process 2 is killed by parent!
```

父进程等待两个子进程终止后,输出如下的信息后终止:

```
parent process is killed!
```

（2）在上面的程序中增加语句 signal（SIGINT，SIG_IGN）和 signal（SIGQUIT，SIG_IGN），观察执行结果，并分析原因。

4. 进程的管道通信

编制一段程序，实现进程的管道通信。

使用系统调用 pipe()建立一条管道线；两个子进程 P1 和 P2 分别向管道各写一句话：

```
Child 1 is sending a message!
Child 2 is sending a message!
```

而父进程则从管道中读出来自两个子进程的信息，显示在屏幕上。

要求父进程先接收子进程 P1 发来的消息，然后再接收子进程 P2 发来的消息。

思　　考

（1）系统是怎样创建进程的？

（2）可执行文件加载时进行了哪些处理？

（3）当首次调用新创建进程时，其入口在哪里？

（4）进程通信有什么特点？

实验 2　进程间通信

实 验 目 的

Linux 系统的进程通信机构(IPC)允许在任意进程间大批量地交换数据。本实验的目的是了解和熟悉 Linux 支持的消息通信机制、共享存储区机制及信息量机制。

实验预备内容

阅读 Linux 系统的 msg.c、sem.c 和 shm.c 等源码文件,熟悉 Linux 的 3 种通信机制。

实 验 内 容

1. 消息的创建、发送和接收

(1) 使用系统调用 msgget()、msgsnd()、msgrcv()及 msgctl(),编制一个长度为 1KB 的消息的发送和接收程序。

(2) 观察上面的程序,说明控制消息队列系统调用 msgctl()在此起什么作用。

2. 共享存储区的创建、附接和断接

使用系统调用 shmget()、shmat()、shmdt()和 shmctl(),编制一个与上述功能相同的程序。

3. 两种消息通信机制的比较

比较上述两种消息通信机制中数据传输的时间。

实验3　存 储 管 理

实 验 目 的

存储管理的主要功能之一是合理地分配空间。请求页式存储管理是一种常用的虚拟存储管理技术。

本实验的目的是通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点，掌握请求页式存储管理的页面置换算法。

实 验 内 容

(1) 通过随机数产生一个指令序列，共 320 条指令。指令的地址按下述原则生成：

① 50％的指令是顺序执行的。

② 25％的指令是均匀分布在前地址部分。

③ 25％的指令是均匀分布在后地址部分。

具体的实施方法如下：

① 在[0,319]的指令地址之间随机选取一个起点 m。

② 顺序执行一条指令，即执行地址为 $m+1$ 的指令。

③ 在前地址[0,$m+1$]中随机选取一条指令并执行，该指令的地址为 m′。

④ 顺序执行一条指令，其地址为 $m'+1$。

⑤ 在后地址[$m'+2$,319]中随机选取一条指令并执行。

⑥ 重复步骤①～⑤，直到执行 320 次指令。

(2) 将指令序列变换成为页地址流。设：

① 页面大小为 1KB。

② 用户内存容量为 4～32 页。

③ 用户虚存容量为 32KB。

在用户虚存中，按每页存放 10 条指令排列虚存地址，即 320 条指令在虚存中的存放方式为：

第 0～9 条指令为第 0 页(对应虚存地址为[0,9])；

第 10～19 条指令为第 1 页(对应虚存地址为[10,19])；

……

第 310～319 条指令为第 31 页(对应虚存地址为[310,319])。

按以上方式，用户指令可组成 32 页。

(3) 计算并输出下述各种算法在不同内存容量下的命中率。

① 先进先出的算法(FIFO)；

② 最近最少使用算法(LRR)；

③ 最佳淘汰算法(OPT)：先淘汰最不常用的页地址；

④ 最少访问页面算法(LFR)；

⑤ 最近最不经常使用算法(NUR)。

其中③和④为选择内容。

命中率计算公式如下：

命中率＝1－页面失效次数/页地址流长度

在本实验中，页地址流长度为320，页面失效次数为每次访问相应指令时该指令所对应的页不在内存的次数。

随机数产生办法

Linux或UNIX系统提供了函数srand()和rand()，分别进行初始化并产生随机数。例如，下面的语句可初始化一个随机数：

```
srand();
```

下面的语句可用来产生a[0]与a[1]中的随机数：

```
a[0]=319 * rand()/32767/32767/2+1;
a[1]=a[0] * rand()/32767/32767/2;
```

实验 4　文件系统设计

实 验 目 的

通过一个简单多用户文件系统的设计，加深理解文件系统的内部功能及内部实现。

实 验 内 容

为 Linux 系统设计一个简单的二级文件系统。要求做到以下几点。

(1) 可以实现下列几条命令(至少 4 条)：

login　　用户登录
dir　　列文件目录
create　　创建文件
delete　　删除文件
open　　打开文件
close　　关闭文件
read　　读文件
write　　写文件

(2) 列目录时要列出文件名、物理地址、保护码和文件长度。

(3) 源文件可以进行读写保护。

实 验 提 示

(1) 首先应确定文件系统的数据结构：主目录、子目录及活动文件等。主目录和子目录都以文件的形式存放于磁盘，这样便于查找和修改。

(2) 用户创建的文件可以编号存储于磁盘上，如 file0、file1、file2 等，并以编号作为物理地址，在目录中进行登记。

实验 1 指导

1. 进程的创建

【任务】

编写一段程序，使用系统调用 fork()创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符，父进程显示字符“a”，子进程分别显示字符“b”和“c”。试观察记录屏幕上的显示结果，并分析原因。

【程序】

```
#include <stdio.h>
main()
{
    int p1,p2;
    while((p1=fork())==-1);              /* 创建子进程 p1 */
    if(p1==0)                            /* 子进程创建成功 */
        putchar('b');
    else
    {
        while((p2=fork())==-1);          /* 创建另一个子进程 */
        if(p2==0)                        /* 子进程创建成功 */
            putchar('c');
        else
            putchar('a');                /* 父进程执行 */
    }
}
```

【运行结果】

bca (有时会出现 bac、acb 等)

【分析】

从进程并发执行来看，输出 bac、acb 等情况都有可能。

fork()创建进程所需的时间要多于输出一个字符的时间，因此在主进程创建进程 2 的同时，进程 1 就输出了“b”，而进程 2 和主程序的输出次序是有随机性的，所以会出现上述结果。

2. 进程的控制

【任务】

修改已编写的程序，将每个进程的输出由单个字符改为一句话，再观察程序执行时屏幕上出现的现象，并分析其原因。如果在程序中使用系统调用 lockf()给每个进程加锁，可以实现进程之间的互斥，观察并分析出现的现象。

【程序 1】

```
#include <stdio.h>
main()
{
    int p1,p2,i;
    while((p1=fork())==-1);
    if(p1==0)
        for(i=0;i<50;i++)
            printf("child %d\n",i);
    else
    {
        while((p2=fork())==-1);
        if(p2==0)
            for(i=0;i<50;i++)
                printf("son %d\n",i);
        else
            for(i=0;i<50;i++)
                printf("daughter %d\n",i);
    }
}
```

【运行结果】

```
child…
son…
daughter…
daughter…
```

或

```
child
…son
…child
…son
…daughter
```

等等。

【分析】

由于函数 printf()输出的字符串中间不会被中断,因此,字符串内部的字符顺序输出时不变。但是,由于进程并发执行时的调度顺序和父子进程的抢占处理机问题,输出字符串的顺序随着执行的不同而发生变化,这与打印单字符的结果类似。

【程序 2】

```
#include <stdio.h>
#include <unistd.h>
```

```
main()
{
    int p1,p2,i;
    while((p1=fork())==-1);
    if(p1==0)
    {
        lockf(1,1,0);
        for(i=0;i<50;i++)
            printf("child %d\n",i);
        lockf(1,0,0);
    }
    else
    {
        while((p2=fork())==-1);
        if(p2==0)
        {
            lockf(1,1,0);
            for(i=0;i<50;i++)
                printf("son %d\n",i);
            lockf(1,0,0);
        }
        else
        {
            lockf(1,1,0);
            for(i=0;i<50;i++)
                printf("daughter %d\n",i);
            lockf(1,0,0);
        }
    }
}
```

【运行结果】

大致与未上锁的输出结果相同，也是随着执行时间不同，输出结果的顺序有所不同。

【分析】

因为上述程序执行时，不同进程之间不存在共享临界资源(其中打印机的互斥性已由操作系统保证)问题，所以，加锁与不加锁效果相同。

3. 软中断通信

【任务 1】

编制一段程序，使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上来的中断信号(即按 Del 键)，当捕捉到中断信号后，父进程用系统调用 kill()向两个子进程发出信号，子进程捕捉到信号后，分别输出下列信息后终止：

```
child process 1 is killed by parent!
```

```
child process 2 is killed by parent!
```

父进程等待两个子进程终止后，输出以下信息后终止：

```
parent process is killed!
```

【程序】

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void waiting(),stop();
int wait_mark;

main()
{
      int p1,p2;
      while((p1=fork())==-1);                    /*创建进程 p1*/
      if(p1>0)
      {
            while((p2=fork())==-1);
            if(p2>0)
            {
                  printf("parent\n");
                  wait_mark=1;
                  signal(SIGINT,stop);           /*接收 Del 信号,并转 stop*/
                  waiting(0);
                  kill(p1,16);                   /*向 p1 发中断信号 16*/
                  kill(p2,17);                   /*向 p2 发中断信号 17*/
                  wait(0);                       /*同步*/
                  wait(0);
                  printf("parent process is killed!\n");
                  exit(0);
            }
            else
            {
                  printf("p2\n");
                  wait_mark=1;
                  signal(17,stop);
                  waiting();
                  lockf(stdout,1,0);
                  printf("child process 2 is killed by parent!\n");
                  lockf(stdout,0,0);
                  exit(0);
            }
      }
```

```
        else
        {
            printf("p1\n");
            wait_mark=1;
            signal(16,stop);
            waiting();
            lockf(stdout,1,0);
            printf("child process 1 is killed by parent!\n");
            lockf(stdout,0,0);
            exit(0);
        }
    }

    void waiting()
    {
        while (wait_mark != 0);
    }

    void stop()
    {
        wait_mark=0;
    }
```

【运行结果】

```
child process 1 is killed by parent!
child process 2 is killed by parent!
parent process is killed!
```

【分析】

(1) 上述程序中,使用系统调用 signal()都放在一段程序的前面部分,而不是在其他接收信号处。这是因为 signal()的执行只是为进程指定信号量 16 或 17 的作用,以及分配相应的与 stop()过程链接的指针。从而,signal()函数必须在程序前面部分执行。

(2) 该程序段前面部分用了两个 wait(0),为什么？请读者思考。

(3) 该程序段中每个进程退出时都用了语句 exit(0),为什么？请读者思考。

【任务 2】

在上面任务 1 中,增加语句 signal(SIGINT, SIG_IGN)和语句 signal(SIGQUIT, SIG_IGN),观察执行结果,并分析原因。这里,signal(SIGINT, SIG_IGN)和 signal(SIGQUIT, SIG_IGN)分别为忽略 Del 键信号和忽略中断信号。

【程序】

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
```

```
int pid1,pid2;
int EndFlag=0,pf1=0,pf2=0;


void IntDelete()
{
      kill(pid1,16);
      kill(pid2,17);
      EndFlag=1;
}
void Int1()
{
      printf("child process 1 is killed by parent!");
      exit(0);
}
void Int2()
{
      printf("child process 2 is killed by parent!");
      exit(0);
}

main()
{
      int exitpid;
      signal(SIGINT,SIG_IGN);
      signal(SIGQUIT,SIG_IGN);
      while((pid1=fork())==-1);
      if(pid1==0)
      {
            printf("p1\n");
            signal(SIGUSR1,Int1);
            signal(16,SIG_IGN);
            pause();
            exit(0);
      }
      else
      {
            while((pid2=fork())==-1);
            if(pid2==0)
            {
                  printf("p2\n");
                  signal(SIGUSR2,Int2);
                  signal(17,SIG_IGN);
                  pause();
                  exit(0);
```

```
        }
        else
        {
            printf("parent\n");
            signal(SIGINT,IntDelete);
            waitpid(-1,&exitpid,0);
            printf("parent process is killed!\n");
            exit(0);
        }
    }
}
```

【运行结果】

请读者将上述程序输入计算机后，执行并观察。

【分析】

由于忽略了中断与退出信号，程序会一直保持阻塞状态而无法退出。

4. 进程的管道通信

【任务】

编制一段程序，实现进程的管道通信。使用系统调用 pipe()建立一条管道线。两个子进程 p1 和 p2 分别向管道各写一句话：

```
child 1 is sending a message!
child 2 is sending a message!
```

而父进程则从管道中读出来自两个子进程的信息，显示在屏幕上。

【程序】

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

int pid1,pid2;

main()
{
    int fd[3];
    char OutPipe[100],InPipe[100];
    pipe(fd);
    while((pid1=fork())==-1);
    if(pid1==0)
    {
        printf("p1\n");
        lockf(fd[1],1,0);
        sprintf(OutPipe,"Child 1 process is sending a message!");
        write(fd[1],OutPipe,50);
```

```
            sleep(1);
            lockf(fd[1],0,0);
            exit(0);
        }
        else
        {
            while((pid2=fork())==-1);
            if(pid2==0)
            {
                printf("p2\n");
                lockf(fd[1],1,0);
                sprintf(OutPipe,"Child 2 process is sending a message!");
                write(fd[1],OutPipe,50);
                sleep(1);
                lockf(fd[1],0,0);
                exit(0);
            }
            else
            {
                printf("parent\n");
                wait(0);
                read(fd[0],InPipe,50);
                printf("%s\n",InPipe);
                wait(0);
                read(fd[0],InPipe,50);
                printf("%s\n",InPipe);
                exit(0);
            }
        }
}
```

【运行结果】

```
Child 1 process is sending a message!
Child 2 process is sending a message!
```

【分析】

请读者自行完成。

实验 2 指导

1. 消息的创建、发送和接收

【任务】

使用系统调用 msgget()、msgsnd()、msgrcv()及 msgctl()，编制一个长度为 1KB 的消息发送和接收的程序。

【程序设计】

(1) 为了便于操作和观察结果，用一个程序作为“引子”，先后 fork()两个子进程 SERVER 和 CLIENT，进行通信。

(2) SERVER 端建立一个 Key 为 75 的消息队列，等待其他进程发来的消息。当遇到类型为 1 的消息，则作为结束信号，取消该队列，并退出 SERVER。SERVER 每接收到一个消息后显示一句“(server)received”。

(3) CLIENT 端使用 key 为 75 的消息队列，先后发送类型从 10 到 1 的消息，然后退出。最后的一个消息即 SERVER 端需要的结束信号。CLIENT 每发送一条消息后显示一句“(client)sent”。

(4) 父进程在 SERVER 和 CLIENT 均退出后结束。

【程序】

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>

#define MSGKEY 75                    /* 定义关键词 MSGKEY */

struct msgform                       /* 消息结构 */
{
    long mtype;
    char mtrex[1030];                /* 文本长度 */
}msg;
int msgqid,i;
void CLIENT()
{
    int i;
    msgqid=msgget(MSGKEY,0777);
    for(i=10;i>=1;i--)
    {
        msg.mtype=i;
        printf("(client)sent\n");
```

```
            msgsnd(msgqid,&msg,1024,0);           /* 发送消息 msg 入 msgqid 消息队列 */
        }
        exit(0);
}

void SERVER()
{
        msgqid=msgget(MSGKEY,0777|IPC_CREAT);     /* 由关键字获得消息队列 */
        do
        {
            msgrcv(msgqid,&msg,1030,0,0);         /* 从 msgqid 队列接收消息 msg */
            printf("(server)received\n");
        }while(msg.mtype!=1);                     /* 消息类型为 1 时,释放队列 */
        msgctl(msgqid,IPC_RMID,0);
        exit(0);
}
void main()
{
        while ((i=fork())==-1);
        if(!i) SERVER();
        while((i=fork())==-1);
        if(!i) CLIENT();
        wait(0);
        wait(0);
}
```

【结果】

从理想的结果来说,应当是每当 CLIENT 发送一个消息后,SERVER 接收该消息,CLIENT 再发送下一条。也就是说“(client)sent”和“(server)received”的字样应该在屏幕上交替出现。实际的结果大多是先由 CLIENT 发送了两条消息,然后 SERVER 接收一条消息,此后 CLIENT_SERVER 交替发送和接收消息,最后 SERVER 一次接收两条消息。CLIENT 和 SERVER 分别发送和接收了 10 条消息,与预期设想一致。

【分析】

message 的传送和控制并不保证完全同步,当一个程序不在激活状态的时候,它完全可能继续睡眠,造成了上面的现象,在多次发送消息后才接收消息。这一点有助于理解消息传送的实现机理。

2. 共享存储区的创建、附接和断接

【任务】

使用系统调用 shmget()、shmat()、shmdt()和 shmctl()编制一个与上述相同功能的程序。

【程序设计】

(1) 为了便于操作和观察结果,用一个程序作为“引子”,先后 fork()两个子进程

SERVER 和 CLIENT ,进行通信。

(2) SERVER 端建立一个 Key 为 75 的共享区,并将第一个字节置为-1。作为数据空的标志。等待其他进程发来的消息。当该字节的值发生变化时,表示收到了信息,进行处理,然后再次把它的值设为-1。如果遇到的值为 0,则视为结束信号,取消该队列,并退出 SERVER。SERVER 每接收到一次数据后显示“(server)received”。

(3) CLIENT 端建立一个 Key 为 75 的共享区,当共享取得第一个字节为-1 时,SERVER 端空闲,可发送请求。CLIENT 随即填入 9 到 0。期间等待 SERVER 端的再次空闲。进行完这些操作后,CLIENT 退出。CLIENT 每发送一次数据后显示“(client)sent”。

(4) 父进程在 SERVER 和 CLIENT 均退出后结束。

【程序】

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>

#define SHMKEY 75                        /* 定义共享区关键词 */

int shmid,i;
int * addr;

void CLIENT()
{
    int i;
    shmid=shmget(SHMKEY,1024,0777);      /* 获取共享区,长度 1024,关键词 SHMKEY */
    addr=shmat(shmid,0,0);               /* 共享区起始地址为 addr */
    for (i=9;i>=0;i--)
    {
        while (* addr!=-1);
        printf("(client)sent\n");        /* 打印 (client)sent */
        * addr=i;                        /* 把 i 赋予 addr */
    }
    exit(0);
}

void SERVER()
{
    shmid=shmget(SHMKEY,1024,0777|IPC_CREAT);   /* 创建共享区 */
    addr=shmat(shmid,0,0);               /* 共享区起始地址为 addr */
    do
    {
        * addr=-1;
        while (* addr==-1);
        printf("(server)received\n");    /* 服务进程使用共享区 */
    }while(* addr);
```

```
        shmctl(shmid,IPC_RMID,0);
        exit(0);
    }
    void main()
    {
        while ((i=fork())==-1);
        if(!i) SERVER();
        while ((i=fork())==-1);
        if(!i) CLIENT();
        wait(0);
        wait(0);
    }
```

【结果】

运行的结果和预想的完全一样。但在运行的过程中,发现每当 CLIENT 发送一次数据后,SERVER 要等待大约 0.1s 才有响应。同样,之后 CLIENT 又需要等待约 0.1s 才发送下一个数据。

【分析】

出现上述的应答延迟现象是程序设计的问题。当 CLIENT 端发送了数据后,并没有任何措施通知 SERVER 端数据已经发出,需要由 CLIENT 的查询才能感知。此时,CLIENT 端并没有放弃系统的控制权,仍然占用 CPU 的时间片。只有当系统进行调度时,切换到了 SERVER 进程,再进行应答。这个问题,也同样存在于 SERVER 端到 CLIENT 的应答过程之中。

3. 比较两种消息通信机制中的数据传输的时间

由于两种机制实现的机理和用处都不一样,难以直接进行时间上的比较。要比较其性能,应更加全面地分析。

(1) 消息队列的建立比共享区的设立消耗的资源少。前者只是一个软件上设定的问题,后者需要对硬件操作,实现内存的映像,当然控制起来比前者复杂。如果每次都重新进行队列或共享的建立,共享区的设立没有什么优势。

(2) 当消息队列和共享区建立好后,共享区的数据传输受到了系统硬件的支持,不耗费多余的资源;而消息传递由软件进行控制和实现,需要消耗一定的 CPU 资源。从这个意义上讲,共享区更适合频繁和大量的数据传输。

(3) 消息的传递自身就带有同步的控制。当等待消息的时候,进程进入睡眠状态,不再消耗 CPU 资源。而共享队列如果不借助其他机制进行同步,接收数据的一方必须进行不断的查询,白白浪费了大量的 CPU 资源。可见,消息方式的使用更加灵活。

实验 3 指导

【任务】

设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。

(1) 先进先出的算法(FIFO)；

(2) 最近最少使用算法(LRU)；

(3) 最佳淘汰算法(OPT)；

(4) 最少访问页面算法(LFU)；

(5) 最近最不经常使用算法(NUR)。

命中率计算公式如下：

命中率＝(1－页面失效次数)/页地址流长度

【程序设计】

本实验的程序设计基本上按照实验内容进行，即首先用 srand()和 rand()函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。相关定义如下。

(1) 数据结构

① 页面类型

```
typedef struct
{
    int pn, pfn,counter,time;
}pl_type;
```

其中 pn 为页号，pfn 为面号，counter 为一个周期内访问该页面的次数，time 为访问时间。

② 页面控制结构

```
pfc_struct
{
    int pn,pfn;
    struct pfc_struct * next;
};
typedef struct pfc_struct pfc_type;
pfc_type pfc[total_vp], * freepf_head, * busypf_head;
pfc_type * busypf_tail;
```

其中 pfc[total_vp]定义用户进程虚页控制结构：

freepf_head 为空页面头的指针。

busypf_head 为忙页面头的指针。

busypf_tail 为忙页面尾的指针。

（2）函数定义

void initialize()：初始化函数，给每个相关的页面赋值。

void FIFO()：计算使用 FIFO 算法时的命中率。

void LRU()：计算使用 LRU 算法时的命中率。

void OPT()：计算使用 OPT 算法时的命中率。

void LFU()：计算使用 LFU 算法时的命中率。

void NUR()：计算使用 NUR 算法时的命中率。

（3）变量定义

int a[tatal_instruction]：指令流数据组。

int page[total_instruction]：每条指令所属页号。

int offset[total_instruction]：每页装入 10 条指令后取模运算页号偏移值。

int total_pf：用户进程的内存页面数。

int diseffect：页面失效次数。

（4）程序流程图

略。

【程序】

```
#define TRUE 1
#define FALSE 0
#define INVALID-1
#define NULL 0

#define total_instruction 320
#define total_vp 32
#define clear_period 50

typedef struct
{
    int pn,pfn,counter,time;
}pl_type;
pl_type pl[32];
typedef struct pfc_struct
{
    int pn,pfn;
    struct pfc_struct *next;
}pfc_type;
pfc_type pfc[32], *freepf_head, *busypf_head, *busypf_tail;

int diseffect, a[total_instruction];
int page[total_instruction], offset[total_instruction];

void initialize();
void FIFO();
```

```
void LRU();
void OPT();
void LFU();
void NUR();

void main()
{
      int s,i,j;
      srand(10*getpid());                      /*由于每次运行时进程号不同,故可用来
                                                  作为初始化随机数队列的“种子”*/
      s=(float)319*rand()/32767/32767/2+1;
      for(i=0;i<total_instruction;i+=4)        /*产生指令队列*/
      {
            if(s<0||s>319)
            {
                  printf("When i==%d,Error,s==%d\n",i,s);
                  exit(0);
            }
            a[i]=s;                            /*任选一个指令访问点 m*/
            a[i+1]=a[i]+1;                     /*顺序执行一条指令*/
            a[i+2]=(float)a[i]*rand()/32767/32767/2;   /*执行前地址指令 m'*/
            a[i+3]=a[i+2]+1;                   /*顺序执行一条指令*/
            s=(float)(318-a[i+2])*rand()/32767/32767/2+a[i+2]+2;
            if((a[i+2]>318)||(s>319))
                  printf("a[%d+2],a number which is:%d and s==%d\n",i,a[i+2],s);
      }
      for (i=0;i<total_instruction;i++)        /*将指令序列变换成页地址流*/
      {
            page[i]=a[i]/10;
            offset[i]=a[i]%10;
      }
      for(i=4;i<=32;i++)                       /*用户内存工作区从4个页面到32个页面*/
      {
            printf("%2d page frames",i);
            FIFO(i);
            LRU(i);
            OPT(i);
            LFU(i);
            NUR(i);
            printf("\n");
      }
}

void initialize(total_pf)                      /*初始化相关数据结构*/
int total_pf;                                  /*用户进程的内存页面数*/
```

```
{
    int i;
    diseffect=0;

    for(i=0;i<total_vp;i++)
    {
        pl[i].pn=i;
        pl[i].pfn=INVALID;                /* 置页面控制结构中的页号,页面为空 */
        pl[i].counter=0;
        pl[i].time=-1;                    /* 页面控制结构中的访问次数为 0,时间为-1 */
    }
    for(i=0;i<total_pf-1;i++)
    {
        pfc[i].next=&pfc[i+1];
        pfc[i].pfn=i;
    }                                     /* 建立 pfc[i-1]和 pfc[i]之间的链接 */
    pfc[total_pf-1].next=NULL;
    pfc[total_pf-1].pfn=total_pf-1;
    freepf_head=&pfc[0];                  /* 空页面队列的头指针为 pfc[0] */
}

void FIFO(total_pf)                       /* 先进先出算法 FIFO(First In First Out) */
int total_pf;                             /* 用户进程的内存页面数 */
{
    int i,j;
    pfc_type *p;
    initialize(total_pf);                 /* 初始化相关页面控制用数据结构 */
    busypf_head=busypf_tail=NULL;         /* 忙页面队列头,队列尾链接 */
    for(i=0;i<total_instruction;i++)
    {
        if(pl[page[i]].pfn==INVALID)      /* 页面失效 */
        {
            diseffect+=1;                 /* 失效次数 */
            if(freepf_head==NULL)         /* 无空闲页面 */
            {
                p=busypf_head->next;
                pl[busypf_head->pn].pfn=INVALID;
                freepf_head=busypf_head;      /* 释放忙页面队列的第一个页面 */
                freepf_head->next=NULL;
                busypf_head=p;
            }
            p=freepf_head->next;              /* 按 FIFO 方式调新页面入内存页面 */
            freepf_head->next=NULL;
            freepf_head->pn=page[i];
            pl[page[i]].pfn=freepf_head->pfn;
```

```
			if(busypf_tail==NULL)
				busypf_head=busypf_tail=freepf_head;
			else
			{
				busypf_tail->next=freepf_head;            /* 空闲页面减少一个 */
				busypf_tail=freepf_head;
			}
			freepf_head=p;
		}
	}
	printf("FIFO:%6.4f ",1-(float)diseffect/320);
}

void LRU(total_pf)                                        /* LRU 算法 */
int total_pf;
{
	int min,minj,i,j,present_time;
	initialize(total_pf);
	present_time=0;

	for(i=0;i<total_instruction;i++)
	{
		if(pl[page[i]].pfn==INVALID)                      /* 页面失效 */
		{
			diseffect++;
			if(freepf_head==NULL)                         /* 无空闲页面 */
			{
				min=32767;
				for(j=0;j<total_vp;j++)                   /* 找出 time 的最小值 */
					if(min>pl[j].time&&pl[j].pfn!=INVALID)
				{
						min=pl[j].time;
						minj=j;
					}
					freepf_head=&pfc[pl[minj].pfn];       /* 腾出一个单元 */
					pl[minj].pfn=INVALID;
					pl[minj].time=-1;
					freepf_head->next=NULL;
			}
			pl[page[i]].pfn=freepf_head->pfn;             /* 有空闲页面,改为有效 */
			pl[page[i]].time=present_time;

			freepf_head=freepf_head->next;                /* 减少一个空闲页面 */
		}
		else
```

```
            pl[page[i]].time=present_time;          /* 命中则增加该单元的访问次数 */
        present_time++;
    }
    printf("LRU:%6.4f ",1-(float)diseffect/320);
}

void NUR(total_pf)                              /* NUR算法 */
int total_pf;
{
    int i,j,dp,cont_flag,old_dp;
    pfc_type *t;

    initialize(total_pf);
    dp=0;
    for(i=0;i<total_instruction;i++)
    {
        if(pl[page[i]].pfn==INVALID)            /* 页面失效 */
        {
            diseffect++;
            if(freepf_head==NULL)               /* 无空闲页面 */
            {
                cont_flag=TRUE;
                old_dp=dp;
                while(cont_flag)
                    if(pl[dp].counter==0&&pl[dp].pfn!=INVALID)
                        cont_flag=FALSE;
                    else
                    {
                        dp++;
                        if(dp==total_vp)
                            dp=0;
                        if(dp==old_dp)
                        {
                            for(j=0;j<total_vp;j++)
                                pl[j].counter=0;
                        }
                    }
                freepf_head=&pfc[pl[dp].pfn];
                pl[dp].pfn=INVALID;
                freepf_head->next=NULL;
            }
            pl[page[i]].pfn=freepf_head->pfn;
            freepf_head=freepf_head->next;
        }
        else
```

```
                pl[page[j]].counter=1;
            if(i%clear_period==0)
                for(j=0;j<total_vp;j++)
                    pl[j].counter=0;
        }
        printf("NUR:%6.4f ",1-(float)diseffect/320);
}

void OPT(total_pf)                                   /* OPT算法 */
int total_pf;
{
        int i,j,max,maxpage,d,dist[total_vp];
        pfc_type *t;
        initialize(total_pf);
        for(i=0;i<total_instruction;i++)
        {
            if(pl[page[i]].pfn==INVALID)
            {
                diseffect++;
                if(freepf_head==NULL)
                {
                    for(j=0;j<total_vp;j++)
                        if(pl[j].pfn!=INVALID)
                            dist[j]=32767;
                        else
                            dist[j]=0;
                        d=1;
                        for(j=i+1;j<total_instruction;j++)
                        {
                            if(pl[page[j]].pfn!=INVALID)
                                dist[page[j]]=d;
                            d++;
                        }
                        max=-1;
                        for(j=0;j<total_vp;j++)
                        if(max<dist[j])
                        {
                        max=dist[j];
                        maxpage=j;
                        }
                        freepf_head=&pfc[pl[maxpage].pfn];
                        freepf_head->next=NULL;
                        pl[maxpage].pfn=INVALID;
                }
                pl[page[i]].pfn=freepf_head->pfn;
```

```
                freepf_head= freepf_head-> next;
            }
        }
        printf("OPT:% 6.4f ",1- (float)diseffect/320);
}

void LFU(total_pf)                                  /* LRU算法 */
int total_pf;
{
        int i,j,min,minpage;
        pfc_type * t;

        initialize(total_pf);
        for(i= 0;i< total_instruction;i+ + )
        {
            if(pl[page[i]].pfn= = INVALID)
            {
                diseffect+ + ;
                if(freepf_head= = NULL)
                {
                    min= 32767;
                    for(j= 0;j< total_vp;j+ + )
                    {
                        if(min> pl[j].counter&&pl[j].pfn!= INVALID)
                        {
                            min= pl[j].counter;
                            minpage= j;
                        }
                        pl[j].counter= 0;
                    }
                    freepf_head= &pfc[pl[minpage].pfn];
                    pl[minpage].pfn= INVALID;
                    freepf_head-> next= NULL;
                }
                pl[page[i]].pfn= freepf_head-> pfn;
                freepf_head= freepf_head-> next;
                pl[page[i]].counter+ + ;
            }
            else
                pl[page[i]].counter+ + ;
        }
        printf("LFU:% 6.4f ",1- (float)diseffect/320);
}
```

【结果】

```
4 page frames FIFO:0.5312 LRU:0.5281 OPT:0.5687 LFU:0.5344 NUR:0.5531
5 page frames FIFO:0.5344 LRU:0.5406 OPT:0.6000 LFU:0.5469 NUR:0.5594
6 page frames FIFO:0.5594 LRU:0.5563 OPT:0.6188 LFU:0.5719 NUR:0.5813
7 page frames FIFO:0.5719 LRU:0.5719 OPT:0.6438 LFU:0.5875 NUR:0.5906
8 page frames FIFO:0.5938 LRU:0.5875 OPT:0.6594 LFU:0.5969 NUR:0.5781
9 page frames FIFO:0.6062 LRU:0.6062 OPT:0.6656 LFU:0.6031 NUR:0.6219
10 page frames FIFO:0.6125 LRU:0.6188 OPT:0.7000 LFU:0.6250 NUR:0.6188
11 page frames FIFO:0.6469 LRU:0.6281 OPT:0.7063 LFU:0.6375 NUR:0.6469
12 page frames FIFO:0.6594 LRU:0.6531 OPT:0.7188 LFU:0.6406 NUR:0.6656
13 page frames FIFO:0.6656 LRU:0.6687 OPT:0.7375 LFU:0.6469 NUR:0.6844
14 page frames FIFO:0.6844 LRU:0.6969 OPT:0.7625 LFU:0.6719 NUR:0.6844
15 page frames FIFO:0.6937 LRU:0.7063 OPT:0.7688 LFU:0.6813 NUR:0.7156
16 page frames FIFO:0.7063 LRU:0.7219 OPT:0.7812 LFU:0.6969 NUR:0.7125
17 page frames FIFO:0.7156 LRU:0.7344 OPT:0.7906 LFU:0.7094 NUR:0.7281
18 page frames FIFO:0.7344 LRU:0.7500 OPT:0.7937 LFU:0.7250 NUR:0.7375
19 page frames FIFO:0.7625 LRU:0.7562 OPT:0.8063 LFU:0.7438 NUR:0.7438
20 page frames FIFO:0.7844 LRU:0.7812 OPT:0.8125 LFU:0.7594 NUR:0.7688
21 page frames FIFO:0.7875 LRU:0.7875 OPT:0.8281 LFU:0.7812 NUR:0.7750
22 page frames FIFO:0.8000 LRU:0.8031 OPT:0.8406 LFU:0.8031 NUR:0.8187
23 page frames FIFO:0.8000 LRU:0.8063 OPT:0.8469 LFU:0.8031 NUR:0.8156
24 page frames FIFO:0.8219 LRU:0.8156 OPT:0.8500 LFU:0.8156 NUR:0.8344
25 page frames FIFO:0.8313 LRU:0.8250 OPT:0.8625 LFU:0.8281 NUR:0.8406
26 page frames FIFO:0.8344 LRU:0.8375 OPT:0.8688 LFU:0.8375 NUR:0.8531
27 page frames FIFO:0.8375 LRU:0.8438 OPT:0.8750 LFU:0.8531 NUR:0.8625
28 page frames FIFO:0.8375 LRU:0.8531 OPT:0.8875 LFU:0.8562 NUR:0.8688
29 page frames FIFO:0.8750 LRU:0.8656 OPT:0.8906 LFU:0.8656 NUR:0.8688
30 page frames FIFO:0.8812 LRU:0.8688 OPT:0.8969 LFU:0.8750 NUR:0.8719
31 page frames FIFO:0.9000 LRU:0.8875 OPT:0.9000 LFU:0.8938 NUR:0.8906
32 page frames FIFO:0.9000 LRU:0.9000 OPT:0.9000 LFU:0.9000 NUR:0.9000
```

【分析】

从上述结果可知，在内存页面数较少（4～5 个页面）时，5 种算法的命中率差别不大，都是 50%左右。在内存页面为 7～25 个时，5 种算法的访内命中率大致在 52%～87%变化。但是，FIFO 算法与 OPT 算法之间的差别一般在 6%～10%。在内存页面为 25～32 个时，由于用户进程的所有指令基本上都已装入内存，命中率已增加较大，从而算法之间的差别不大。

比较上述 5 种算法，以 OPT 算法的命中率最高，NUR 算法次之，随后是 LFU 算法和 LRU 算法，最后是 FIFO 算法。

实验 4 指导

【任务】

为 Linux 系统设计一个简单的二级文件系统。要求做到以下几点。

(1) 可以实现下列几条命令:

login　　用户登录
dir　　列目录
create　　创建文件
delete　　删除文件
open　　打开文件
close　　关闭文件
read　　读文件
write　　写文件

(2) 列目录时要列出文件名、物理地址、保护码和文件长度。

(3) 源文件可以进行读写保护

【程序设计】

(1) 设计思想

本文件系统采用两级目录,其中第一级对应于用户账号,第二级对应于用户账号下的文件。另外,为了简单,本文件系统未考虑文件共享、文件系统安全以及管道文件与设备文件等特殊内容。对这些内容感兴趣的读者,可以在本系统的程序基础上进行扩充。

(2) 主要数据结构

① 索引节点

```
struct inode
{
    struct inode * i_forw;
    struct inode * i_back;
    char i_flag;
    unsigned int i_ino;                /* 磁盘索引节点标号 */
    unsigned int i_count;              /* 引用计数 */
    unsigned short di_number;          /* 关联文件数,当为 0 时,则删除该文件 */
    unsigned short di_mode;            /* 存取权限 */
    unsigned short di_uid;             /* 磁盘索引节点用户 id */
    unsigned short di_gid;             /* 磁盘索引节点组 id */
    unsigned int di_addr [NADDR];      /* 物理块号 */
}
```

② 磁盘索引节点

```
struct dinode
{
    unsigned short di_number;        /* 关联文件数 */
    unsigned short di_mode;          /* 存取权限 */

    unsigned short di_uid;
    unsigned short di_gid;
    unsigned long di_size;           /* 文件大小 */
    unsigned int di_addr[NADDR];     /* 物理块号 */
}
```

③ 目录项结构

```
struct direct
{
    char d_name[DIRSIZ];             /* 目录名 */
    unsigned int d_ino;              /* 目录号 */
}
```

④ 超级块

```
struct filsys
{
    unsigned short s_isize;          /* 索引节点块块数 */
    unsigned long s_fsize;           /* 数据块块数 */

    unsigned int s_nfree;            /* 空闲块块数 */
    unsigned short s_pfree;          /* 空闲块指针 */
    unsigned int s_free [NICFREE];   /* 空闲块堆栈 */

    unsigned int s_ninode;           /* 空闲索引节点数 */
    unsigned short s_pinode;         /* 空闲索引节点指针 */
    unsigned int s_inode [NICINOD];  /* 空闲索引节点数组 */
    unsigned int s_rinode;           /* 铭记索引节点 */

    char s_fmod;                     /* 超级块修改标志 */

};
```

⑤ 用户密码

```
struct pwd
{
    unsigned short p_uid;
    unsigned short p_gid;
    char password [PWOSIZ];
```

```
};
```

⑥ 目录

```
struct dir
{
      struct direct direct [DIRNUM];
      int size;
};
```

⑦ 查找内存索引节点的 hash 表

```
struct hinode
{
      struct inode * i_forw;
};
```

⑧ 系统打开表

```
struct file
{
      char f_flag;                          /* 文件操作标志 */
      unsigned int f_count;                 /* 引用计数 */
      struct inode * f_inode;               /* 指向内存索引节点 */
      unsigned long f_off;                  /* 读/写指针 */
};
```

⑨ 用户打开表

```
struct user
{
      unsigned short u_default_mode;
      unsigned short u_uid;                 /* 用户标志 */
      unsigned short u_gid;                 /* 用户组标志 */
      unsigned short u_ofile [NOFILE];      /* 用户打开文件表 */
};
```

（3）主要函数

① iget()　　索引节点内容获取函数

② iput()　　索引节点内容释放函数

③ mkdir()　　目录创建函数

④ namei()　　目录搜索函数

⑤ balloc()　　磁盘块分配函数

⑥ bfree()　　磁盘块释放函数

⑦ ialloc()　　分配索引节点区函数

⑧ ifree()　　释放索引节点区函数

⑨ iname()　　搜索当前目录下文件的函数

⑩ access()　　访问控制函数

⑪ _dir()　　　显示目录和文件用函数
⑫ chdir()　　　改变当前目录用函数
⑬ open()　　　打开文件函数
⑭ create()　　　创建文件函数
⑮ read()　　　读文件用函数
⑯ write()　　　写文件用函数
⑰ login()　　　用户登录函数
⑱ logout()　　　用户退出函数
⑲ format()　　　文件系统格式化函数
⑳ install()　　　进入文件系统函数
㉑ close()　　　关闭文件函数
㉒ halt()　　　退出文件系统函数
㉓ delete()　　　文件删除函数

以上函数的详细描述略。

(4) 主程序说明

```
Begin
    Step1 对磁盘进行格式化
    Step2 调用 install(),进入文件系统
    Step3 调用_dir(),显示当前目录
    Step4 调用 login(),用户注册
    Step5 调用 mkdir()和 chdir()创建目录
    Step6 调用 create(),创建文件 0
    Step7 分配缓冲区
    Step8 写文件 0
    Step9 关闭文件 0并释放缓冲区
    Step10 调用 mkdir()和 chdir()创建子目录
    Step11 调用 create(),创建文件 1
    Step12 分配缓冲区
    Step13 写文件 1
    Step14 关闭文件 1并释放缓冲区
    Step15 调用 chdir 将当前目录移到上一级
    Step16 调用 create(),创建文件 2
    Step17 分配缓冲区
    Step18 调用 write(),写文件 2
    Step19 关闭文件 2并释放缓冲区
    Step20 调用 delete(),删除文件 0
    Step21 调用 create(),创建文件 3
    Step22 为文件 3分配缓冲区
    Step23 调用 write(),写文件 3
    Step24 关闭文件 3并释放缓冲区
    Step25 调用 open(),打开文件 2
    Step26 为文件 2分配缓冲区
```

```
        Step27 写文件 3后关闭文件 3
        Step28 释放缓冲区
        Step29 用户退出 (logout)
        Step30 关闭 (halt)
End
```

由上述描述过程可知，该文件系统实际是为用户提供一个解释执行相关命令的环境。主程序中的大部分语句都被用来执行相应的命令。

下面给出每个过程的相关 C 语言程序。读者也可以使用这些子过程，编写出一个用 Shell 控制的文件系统界面。

【程序】

（1）编程管理文件 makefile

本文件系统程序用 makefile 编程管理工具进行管理。其内容如下：

```
/* makefile */
filsys: main.o igetput.o iallfre.o ballfre.o name.o access.o log.o close.o
        create.o delete.o dir.o dirlt.o open.o rdwt.o format.o install.o halt.o
cc-o filsys main.o igetput.o iallfre.o ballfre.o name.o access.o log.o close.o
        create.o delete.o dir.o dirlt.o open.o rdwt.o format.o install.o halt.o
main.o: main.c filesys.h
        cc-c main.c
igetput.o: igetput.c filesys.h
        cc-c igetput.c
iallfre.o: iallfre.c filesys.h
        cc-c iallfre.c
ballfre.o: ballfre.c filesys.h
        cc-c ballfre.c
name.o: name.c filesys.h
        cc-c name.c
access.o: access.c filesys.h
        cc-c access.c
log.o: log.c filesys.h
        cc-c log.c
close.o: close.c filesys.h
        cc-c close.c
create.o: create.c filesys.h
        cc-c create.c
delete.o: delete.c filesys.h
        cc-c delete.c
dir.o: dir.c filesys.h
        cc-c dir.c
dirlt.o: dirlt.c filesys.h
        cc-c dirlt.c
open.o: open.c filesys.h
        cc-c open.c
```

```
rdwt.o: rdwt.c filesys.h
      cc- c rdwt.c
format.o: format.c filesys.h
      cc- c format.c
install.o: install.c filesys.h
      cc- c install.c
halt.o: halt.c
      cc- c halt.c
```

(2) 头文件 filesys.h

头文件 filesys.h 用来定义本文件系统中所使用的各种数据结构和常数。

```
/* filesys.h定义本文件系统中的数据结构和常数 */
#define BLOCKSIZ 512
#define SYSOPENFILE 40
#define DIRNUM 128
#define DIRSIZ 14
#define PWDSIZ 12
#define PWDNUM 32
#define NOFILE 20
#define NADDR 10
#define NHINO 128
#define USERNUM 10
#define DINODESIZ 32

/* filsys */
#define DINODEBLK 32
#define FILEBLK 512
#define NICFREE 50
#define NICINOD 50
#define DINODESTART 2* BLOCKSIZ
#define DATASTART (2+ DINODEBLK) * BLOCKSIZ

/* di_mode */
#define DIEMPTY 0000

#define DIFILE 01000
#define DIDIR 02000

#define UDIREAD 00001 /* user */
#define UDIWRITE 00002
#define UDIEXICUTE 00004
#define GDIREAD 00010 /* group */
#define GDIWRITE 00020
#define GDIEXICUTE 00040
#define ODIREAD 00100 /* other */
```

```
#define ODIWRITE 00200
#define ODIEXICUTE 00400

#define READ 1
#define WRITE 2
#define EXICUTE 4

#define DEFAULTMODE 00777

/* i_flag */
#define IUPDATE 00002

/* s_fmod */
#define SUPDATE 00001

/* f_flag */
#define FREAD 00001
#define FWRITE 00002
#define FAPPEND 00004

/* error */
#define DISKFULL 65535

/* fseek origin */
#define SEEK_SET 0

/* 文件系统数据结构 */
struct inode{
    struct inode * i_forw;
    struct inode * i_back;
    char i_flag;
    unsigned int i_ino;                 /* 磁盘索引节点标志 */
    unsigned int i_count;               /* 引用计数 */
    unsigned short di_number;           /* 关联文件数,当为 0 时,则删除该文件 */
    unsigned short di_mode;             /* 存取权限 */
    unsigned short di_uid;
    unsigned short di_gid;
    unsigned int di_size;               /* 文件大小 */
    unsigned int di_addr[NADDR];        /* 物理块号 */
};

struct dinode{
    unsigned short di_number;           /* 关联文件数 */
    unsigned short di_mode;             /* 存取权限 */
    unsigned short di_uid;
```

```
    unsigned short di_gid;
    unsigned long di_size;                  /* 文件大小 */
    unsigned int di_addr[NADDR];            /* 物理块号 */
};

struct direct{
    char d_name[DIRSIZ];
    unsigned int d_ino;
};

struct filsys{
    unsigned short s_isize;                 /* 索引节点块块数 */
    unsigned long s_fsize;                  /* 数据块块数 */
    unsigned int s_nfree;                   /* 空闲块块数 */
    unsigned short s_pfree;                 /* 空闲块指针 */
    unsigned int s_free[NICFREE];           /* 空闲块堆栈 */

    unsigned int s_ninode;                  /* 空闲索引节点数 */
    unsigned short s_pinode;                /* 空闲索引节点指针 */
    unsigned int s_inode[NICINOD];          /* 空闲索引节点数组 */
    unsigned int s_rinode;                  /* 铭记索引节点 */

    char s_fmod;                            /* 超级块修改标志 */
};

struct pwd{
    unsigned short p_uid;
    unsigned short p_gid;

    char password[PWDSIZ];
};
struct dir{
    struct direct direct[DIRNUM];
    int size;                               /* 当前目录大小 */
};

struct hinode{
    struct inode * i_forw;                  /* hash表指针 */
};

struct file{
    char f_flag;                            /* 文件操作标志 */
    unsigned int f_count;                   /* 引用计数 */

    struct inode * f_inode;                 /* 指向内存索引节点 */
```

```
    unsigned long f_off;                    /* 读/写指针 */
};

struct user{
    unsigned short u_default_mode;
    unsigned short u_uid;
    unsigned short u_gid;
    unsigned short u_ofile[NOFILE];         /* 用户打开文件表 */
};

/* 以下为全局变量 */
extern struct hinode hinode[NHINO];
extern struct dir dir;                      /* 当前目录(在内存中全部读入) */
extern struct file sys_ofile[SYSOPENFILE];
extern struct filsys filsys;                /* 内存中的超级块 */
extern struct pwd pwd[PWDNUM];
extern struct user user[USERNUM];
extern FILE * fd;     /* the file system colum of all the system */
extern struct inode * cur_path_inode;
extern int user_id,file_block;

/* proptype of the sub routine used in the file system */
extern struct inode * iget();
extern iput();
extern unsigned int balloc();
extern bfree();
extern struct inode * ialloc();
extern ifree();
extern unsigned int namei();
extern unsigned int iname();
extern unsigned int access();
extern _dir();
extern mkdir();
extern chdir();
extern dirlt();
extern unsigned short open();
extern create();
extern unsigned int read();
extern unsigned write();
extern int login();
extern logout();
extern install();
extern format();
extern close();
extern halt();
```

（3）主程序 main()（文件名 main. c）

主程序 main. c 用来测试文件系统的各种设计功能，其主要功能描述如程序设计中的第 4 部分。

程序：

```
#include <string.h>
#include <stdio.h>
#include "filesys.h"

struct hinode hinode[NHINO];

struct dir dir;
struct file sys_ofile[SYSOPENFILE];
struct filsys filsys;
struct pwd pwd[PWDNUM];
struct user user[USERNUM];
FILE * fd;
struct inode * cur_path_inode;
int user_id,file_block;

main()
{
    unsigned short ab_fd1,ab_fd2,ab_fd3,ab_fd4;
    unsigned short bhy_fd1;
    char * buf;

    printf("\nDo you want to format the disk?\n");
    if (getchar()== 'y')
    {
        printf("Format will erase all context on the disk.Are you sure?\n");
        getchar();
    }
    else
        return;
    if(getchar()== 'y')
        format();
    else
        return;

    install();
    _dir();
    login(2118,"abcd");
    user_id= 0;

    mkdir("a2118");
```

```
chdir("a2118");
ab_fd1=create(user_id,"file0.c",01777);
file_block=BLOCKSIZ*6+5;
buf=(char*)malloc(BLOCKSIZ*6+5);
write(ab_fd1,buf,BLOCKSIZ*6+5);
close(user_id,ab_fd1);
free(buf);

mkdir("subdir");
chdir("subdir");
ab_fd2=create(user_id,"file1.c",01777);
file_block=BLOCKSIZ*4+20;
buf=(char*)malloc(BLOCKSIZ*4+20);
write(ab_fd2,buf,BLOCKSIZ*4+20);
close(user_id,ab_fd2);
free(buf);

chdir("..");
ab_fd3=create(user_id,"file2.c",01777);
file_block=BLOCKSIZ*3+255;
buf=(char*)malloc(BLOCKSIZ*3+255);
write(ab_fd3,buf,BLOCKSIZ*3+255);
close(user_id,ab_fd3);
free(buf);

_dir();
delete("ab_file0.c");

ab_fd4=creat(user_id,"file3.c",01777);
file_block=BLOCKSIZ*8+300;
buf=(char*)malloc(BLOCKSIZ*8+300);
write(ab_fd4,buf,BLOCKSIZ*8+300);
close(user_id,ab_fd4);
free(buf);

_dir();
ab_fd3=open(user_id,"file2.c",FAPPEND);
file_block=BLOCKSIZ*3+100;
buf=(char*)malloc(BLOCKSIZ*3+100);
write(ab_fd3,buf,BLOCKSIZ*3+100);
close(user_id,ab_fd3);
free(buf);

_dir();
chdir("..");
```

```
        logout();
        halt();
}
```

(4) 初始化磁盘格式程序 format()(文件名 format.c)

```
#include <stdio.h>
#include "filesys.h"

format()
{
        struct inode * inode;
        struct direct dir_buf[BLOCKSIZ/(DIRSIZ+2)];

        struct filsys;
        unsigned int block_buf[BLOCKSIZ/sizeof(int)];
        char * buf;
        int i,j;

        /* create the file system */
        fd=fopen("filesystem","r+w+b");
        buf=(char *)malloc((DINODEBLK+FILEBLK+2) * BLOCKSIZ * sizeof(char));
        if(buf==NULL)
        {
                printf("\nfile system file create failed!\n");
                exit(0);
        }
        fseek(fd,0,SEEK_SET);
        fwrite(buf,1,(DINODEBLK+FILEBLK+2) * BLOCKSIZ * sizeof(char),fd);

        /* 0.initialize the password */
        pwd[0].p_uid=2116;
        pwd[0].p_gid=03;
        strcpy(pwd[0].password,"dddd");
        pwd[1].p_uid=2117;
        pwd[1].p_gid=03;
        strcpy(pwd[1].password,"bbbb");
        pwd[2].p_uid=2118;
        pwd[2].p_gid=04;
        strcpy(pwd[2].password,"abcd");
        pwd[3].p_uid=2119;
        pwd[3].p_gid=04;
        strcpy(pwd[3].password,"cccc");
        pwd[4].p_uid=2220;
        pwd[4].p_gid=05;
        strcpy(pwd[4].password,"eeee");
```

```
/* 1. create the main directory and its sub dir etc and the file password */
inode=iget(0);                    /* 0 empty dinode id */
inode->di_mode=DIEMPTY;
iput(inode);

inode=iget(1);                    /* 1 main directory id */
inode->di_number=1;
inode->di_mode=DEFAULTMODE|DIDIR;
inode->di_size=3*(DIRSIZ+2);
inode->di_addr[0]=0;              /* block 0# is used by the main directory */
strcpy(dir_buf[0].d_name,"..");
dir_buf[0].d_ino=1;
strcpy(dir_buf[1].d_name,".");
dir_buf[1].d_ino=1;
strcpy(dir_buf[2].d_name,"etc");
dir_buf[2].d_ino=2;
fseek(fd,DATASTART,SEEK_SET);
fwrite(dir_buf,1,3*(DIRSIZ+2),fd);
iput(inode);

inode=iget(2);                    /* 2 etc dir id */
inode->di_number=1;
inode->di_mode=DEFAULTMODE | DIDIR;
inode->di_size=3*(DIRSIZ+2);
inode->di_addr[0]=1;              /* block 0# is used by the etc directory */
strcpy(dir_buf[0].d_name,"..");
dir_buf[0].d_ino=1;
strcpy(dir_buf[1].d_name,".");
dir_buf[1].d_ino=2;
strcpy(dir_buf[2].d_name,"password");
dir_buf[2].d_ino=3;
fseek(fd,DATASTART+BLOCKSIZ*1,SEEK_SET);
fwrite(dir_buf,1,3*(DIRSIZ+2),fd);
iput(inode);

inode=iget(3);                    /* 3 password id */
inode->di_number=1;
inode->di_mode=DEFAULTMODE | DIDIR;
inode->di_size=BLOCKSIZ;
inode->di_addr[0]=2;              /* block 2# is used by the password file */
for(i=5;i<PWDNUM;i++)
{
    pwd[i].p_uid=0;
    pwd[i].p_gid=0;
```

```
        strcpy(pwd[i].password,"      ");
    }
    fseek(fd,DATASTART+2*BLOCKSIZ,SEEK_SET);
    fwrite(pwd,1,BLOCKSIZ,fd);
    iput(inode);

    /* 2. initialize the superblock */
    filsys.s_isize=DINODEBLK;
    filsys.s_fsize=FILEBLK;

    filsys.s_ninode=DINODEBLK*BLOCKSIZ/DINODESIZ-4;
    filsys.s_nfree=FILEBLK-3;

    for(i=0;i<NICINOD;i++)
    {
        /* begin with 4,0,1,2,3, is used by main,etc, password */
        filsys.s_inode[i]=4+i;
    }

    filsys.s_pinode=0;
    filsys.s_rinode=NICINOD+4;

    block_buf[NICFREE-1]=FILEBLK+1;     /* FILEBLK+1 is a flag of end */
    for(i=0;i<NICFREE-1;i++)
        block_buf[NICFREE-2-i]=FILEBLK-i;
    fseek(fd,DATASTART+BLOCKSIZ*(FILEBLK-NICFREE-1),SEEK_SET);
    fwrite(block_buf,1,BLOCKSIZ,fd);

    for(i=FILEBLK-NICFREE-1;i>2;i-=NICFREE)
    {
        for(j=0;j<NICFREE;j++)
        {
            block_buf[j]=i-j;
        }
        block_buf[j]=50;
        fseek(fd,DATASTART+BLOCKSIZ*(i-1),SEEK_SET);
        fwrite(block_buf,1,BLOCKSIZ,fd);
    }

    j=i+NICFREE;
    for(i=j;i>2;i--)
    {
        filsys.s_free[NICFREE-1+i-j]=i;
    }
```

```
        filsys.s_pfree=NICFREE-1-j+3;
        filsys.s_pinode=0;

        fseek(fd,BLOCKSIZ,SEEK_SET);
        fwrite(&filsys,1,sizeof(filsys),fd);
        fseek(fd,BLOCKSIZ,SEEK_SET);
        fread(&filsys.s_isize,1,sizeof(filsys),fd);
}
```

(5) 进入文件系统程序 install()(文件名 install.c)

```
#include <stdio.h>
#include <string.h>
#include "filesys.h"

install()
{
        int i,j;

        /* 1.read the filsys from the superblock */
        fseek(fd,BLOCKSIZ,SEEK_SET);
        fread(&filsys,1,sizeof(struct filsys),fd);

        /* 2. initialize the inode hash chain */
        for(i=0;i<NHINO;i++)
        {
                hinode[i].i_forw=NULL;
        }

        /* 3.initialize the sys_ofile */
        for(i=0;i<SYSOPENFILE;i++)
        {
                sys_ofile[i].f_count=0;
                sys_ofile[i].f_inode=NULL;
        }

        /* 4.initialize the user */
        for(i=0;i<USERNUM;i++)
        {
                user[i].u_uid=0;
                user[i].u_gid=0;
                for(j=0;j<NOFILE;j++)
                        user[i].u_ofile[j]=SYSOPENFILE+1;
        }

        /* 5.read the main directory to initialize the dir */
```

```
    cur_path_inode= iget(1);
    dir.size= cur_path_inode-> di_size/(DIRSIZ+ 2);

    for(i= 0;i< DIRNUM;i++)
    {
        strcpy(dir.direct[i].d_name,"      ");
        dir.direct[i].d_ino= 0;
    }

    for(i= 0;i< dir.size/(BLOCKSIZ/(DIRSIZ+ 2));i++)
    {
        fseek(fd,DATASTART+ BLOCKSIZ * cur_path_inode-> di_addr[i],SEEK_SET);
        fread(&dir.direct[(BLOCKSIZ/(DIRSIZ+ 2)) * i],1,BLOCKSIZ,fd);
    }

    fseek(fd,DATASTART+ BLOCKSIZ * cur_path_inode-> di_addr[i],SEEK_SET);
    fread(&dir.direct[(BLOCKSIZ)/(DIRSIZ+ 2) * i],1,cur_path_inode-> di_size% BLOCKSIZ,fd);
}
```

(6) 退出程序 halt()(文件名 halt.c)

```
#include <stdio.h>
#include "filesys.h"

halt()
{
    struct inode * inode;
    int i,j;

    /* 1. write back the current dir */
    chdir("..");
    iput(cur_path_inode);

    /* 2. free the u_ofile and sys_ofile and inode */
    for(i= 0;i< USERNUM;i++)
    {
        if(user[i].u_uid!= 0)
        {
            for(j= 0;j< NOFILE;j++)
            {
                if(user[i].u_ofile[j]!= SYSOPENFILE+ 1)
                {
                    close(i,j);
                    user[i].u_ofile[j]= SYSOPENFILE+ 1;
                }
            }
```

```
        }
    }

    /* 3. write back the filsys to the disk */
    fseek(fd,BLOCKSIZ,SEEK_SET);
    fwrite(&filsys,1,sizeof(struct filsys),fd);

    /* 4. close the file system colum */
    fclose(fd);

    /* 5. say GOODBYE to all the user */
    printf("\nGoodbye. See you Next Time. Please turn off the switch\n");
    exit(0);
}
```

(7) 获取释放索引节点内容程序 iget()/iput()(文件名 igetput.c)

```
#include <stdio.h>
#include "filesys.h"

struct inode * iget(dinodeid)                        /* iget() */
unsigned int dinodeid;
{
    int existed=0,inodeid;
    long addr;
    struct inode * temp, * newinode;

    inodeid=dinodeid%NHINO;
    if(hinode[inodeid].i_forw==NULL)
        existed=0;
    else
    {
        temp=hinode[inodeid].i_forw;
        while(temp)
        {
            if(temp->i_ino==inodeid)
            /* existed */
            {
                existed=1;
                temp->i_count++;
                return temp;
            }
            /* not existed */
            else
                temp=temp->i_forw;
        }
```

```
        }

        /* not existed */
        /* 1. calculate the addr of the dinode in the file sys column */
        addr=DINODESTART+dinodeid*DINODESIZ;

        /* 2.malloc the new inode */
        newinode=(struct inode *)malloc(sizeof(struct inode));

        /* 3. read the dinode to the inode */
        fseek(fd,addr,SEEK_SET);
        fread(&(newinode->di_number),DINODESIZ,1,fd);

        /* 4. put it into hinode[inodeid] queue */
        newinode->i_forw=hinode[inodeid].i_forw;
        newinode->i_back=newinode;
        if(newinode->i_forw!=NULL)
            newinode->i_forw->i_back=newinode;
        hinode[inodeid].i_forw=newinode;

        /* 5. initialize the inode */
        newinode->i_count=1;
        newinode->i_flag=0;
        newinode->i_ino=dinodeid;

        newinode->di_size=3*(DIRSIZ+2);
        if(dinodeid==3)
              newinode->di_size=BLOCKSIZ;
        return newinode;
}

iput(pinode)              /* iput() */
struct inode *pinode;
{
        long addr;
        unsigned int block_num;
        int i;

        if(pinode->i_count>1)
        {
              pinode->i_count--;
              return;
        }
        else
        {
```

```
            if(pinode->di_number!=0)
            {
                /* write back the inode */
                addr=DINODESTART+pinode->i_ino*DINODESIZ;
                fseek(fd,addr,SEEK_SET);
                fwrite(&pinode->di_number,DINODESIZ,1,fd);
            }
            else
            {
                /* rm the inode & the block of the file in the disk */
                block_num=pinode->di_size/BLOCKSIZ;
                for(i=0;i<block_num;i++)
                    bfree(pinode->di_addr[i]);
                ifree(pinode->i_ino);
            }

            /* free the inode in the memory */
            if(pinode->i_forw==NULL)
                pinode->i_back->i_forw=NULL;
            else
            {
                pinode->i_forw->i_back=pinode->i_back;
                pinode->i_back->i_forw=pinode->i_forw;
            }
            ifree(pinode);
        }
}
```

(8) 索引节点分配和释放函数 ialloc()和 ifree()(文件名 iallfre.c)

```
#include <stdio.h>
#include "filesys.h"

static struct dinode block_buf[BLOCKSIZ/DINODESIZ];

struct inode *ialloc()                    /* ialloc */
{
    struct inode *temp_inode;
    unsigned int cur_di;
    int i,count,block_end_flag;

    if(filsys.s_pinode==NICINOD)          /* s_inode empty */
    {
        i=0;
        count=0;
        block_end_flag=1;
```

```
        filsys.s_pinode=NICINOD-1;
        cur_di=filsys.s_rinode;
        while((count<NICINOD)||(count<=filsys.s_ninode))
        {
            if(block_end_flag)
            {
                fseek(fd,DINODESTART+cur_di*DINODESIZ,SEEK_SET);
                fread(block_buf,1,BLOCKSIZ,fd);
                block_end_flag=0;
                i=0;
            }
            while(block_buf[i].di_mode==DIEMPTY)
            {
                cur_di++;
                i++;
            }
            if(i==NICINOD)
                block_end_flag=1;
            else
            {
                filsys.s_inode[filsys.s_pinode--]=cur_di;
                count++;
            }
        }
        filsys.s_rinode=cur_di;
    }
    temp_inode=iget(filsys.s_inode[filsys.s_pinode]);
    fseek(fd,DINODESTART+filsys.s_inode[filsys.s_pinode]*DINODESIZ,SEEK_SET);
    fwrite(&temp_inode->di_number,1,sizeof(struct dinode),fd);
    filsys.s_pinode++;
    filsys.s_ninode--;
    filsys.s_fmod=SUPDATE;
    return temp_inode;
}

ifree(dinodeid)                             /*ifree*/
unsigned dinodeid;
{
    filsys.s_ninode++;
    if(filsys.s_pinode!=NICINOD)            /*not full*/
    {
        filsys.s_inode[filsys.s_pinode]=dinodeid;
        filsys.s_pinode++;
    }
    else                                    /*full*/
```

```
	{
		if(dinodeid< filsys.s_rinode)
		{
			filsys.s_inode[NICINOD]= dinodeid;
			filsys.s_rinode= dinodeid;
		}
	}
}
```

(9) 磁盘块分配与释放函数 balloc()与 bfree()(文件名 ballfre.c)

```
#include < stdio.h>
#include "filesys.h"

static unsigned int block_buf[BLOCKSIZ];

unsigned int balloc()
{
	unsigned int free_block,free_block_num;
	int i;

	if(filsys.s_nfree= = 0)
	{
		printf("\nDisk Full!\n");
		return DISKFULL;
	}

	free_block= filsys.s_free[filsys.s_pfree];
	if(filsys.s_pfree= = NICFREE- 1)
	{
		fseek(fd,DATASTART+ (562- filsys.s_nfree) * BLOCKSIZ,SEEK_SET);
		fread(block_buf,1,BLOCKSIZ,fd);
		free_block_num= block_buf[NICFREE];                /* the total number in the group */
		for(i= 0;i< free_block_num;i+ + )
		{
			filsys.s_free[NICFREE- 1- i]= block_buf[i];
		}
		filsys.s_pfree= NICFREE- free_block_num;
	}
	else
		filsys.s_pfree+ + ;

	filsys.s_nfree- - ;
	filsys.s_fmod= SUPDATE;

		return free_block;
```

```
        }

        bfree(block_num)
        unsigned int block_num;
        {
                int i;

                if(filsys.s_free==0)                                            /* s_free full */
                {
                        block_buf[NICFREE]=NICFREE;
                        for(i=0;i<NICFREE;i++)
                        {
                                block_buf[i]=filsys.s_free[NICFREE-1-i];
                        }
                        filsys.s_pfree=NICFREE-1;
        }

        fseek(fd,BLOCKSIZ,SEEK_SET);
        fwrite(block_buf,1,BLOCKSIZ,fd);
        filsys.s_nfree++;
        filsys.s_fmod=SUPDATE;
}
```

(10) 搜索函数 namei()和 iname()(文件名 name.c)

```
#include <string.h>
#include <stdio.h>
#include "filesys.h"

unsigned int namei(name)                        /* namei */
char *name;
{
        int i,notfound=1;

        for(i=0;((i<dir.size)&&(notfound));i++)
                if((!strcmp(dir.direct[i].d_name,name))&&(dir.direct[i].d_ino!=0))
                        return dir.direct[i].d_ino;             /* find */
        return 0;                                       /* not find */
}

unsigned int iname(name)                        /* iname */
char *name;
{
        int i,notfound=1;

        for(i=0;((i<DIRNUM)&&(notfound));i++)
```

```
            if(dir.direct[i].d_ino==0)
            {
                notfound=0;
                break;
            }

    if(notfound)
    {
        printf("\n The current directory is full!\n");
        return 0;
    }
    else
    {
        strcpy(dir.direct[i].d_name,name);
        dir.direct[i].d_ino=1;
        return i;
    }
}
```

(11) 访问控制函数 access()(文件名 access.c)

```
#include <stdio.h>
#include "filesys.h"

unsigned int access(user_id,inode,mode)
unsigned int user_id;
struct inode * inode;
unsigned short mode;
{
    switch (mode)
    {
        case READ:
            if(inode->di_mode & ODIREAD) return 1;
            if((inode->di_mode & GDIREAD)
                &&(user[user_id].u_gid==inode->di_gid)) return 1;
            if((inode->di_mode & UDIREAD)
                && (user[user_id].u_uid==inode->di_uid)) return 1;
            return 0;
        case WRITE:
            if(inode->di_mode & ODIWRITE) return 1;
            if((inode->di_mode & GDIWRITE)
                && (user[user_id].u_gid==inode->di_gid)) return 1;
            if((inode->di_mode& UDIWRITE)
                &&(user[user_id].u_uid==inode->di_uid)) return 1;
            return 0;
        case EXICUTE:
```

```
            if(inode->di_mode & ODIEXICUTE) return 1;
            if((inode->di_mode & GDIEXICUTE)
                && (user[user_id].u_gid= = inode->di_gid)) return 1;
            if((inode->di_mode & UDIEXICUTE)
                &&(user[user_id].u_uid= = inode->di_uid)) return 1;
            return 0;
        case DEFAULTMODE:
            return 1;
        default:
            return 0;
    }
}
```

(12) 显示列表函数_dir()和目录创建函数 mkdir()等(文件名 dir.c)

```
#include<stdio.h>
#include<string.h>
#include "filesys.h"

_dir() /* _dir */
{
    unsigned short di_mode;
    int i,j,k,one;
    struct inode *temp_inode;

    printf("\nCURRENT DIRECTORY:dir.size=%d\n",dir.size);
    for(i=0;i<dir.size;i++)
    {
        if(dir.direct[i].d_ino!=DIEMPTY)
        {
            printf("%20s ",dir.direct[i].d_name);
            temp_inode=iget(dir.direct[i].d_ino);
            di_mode=temp_inode->di_mode;

            if(temp_inode->di_mode & DIFILE)
                printf("f");
            else
                printf("d");

            for(j=0;j<9;j++)
            {
                one=di_mode%2;
                di_mode=di_mode/2;
                if(one)
                    printf("x");
                else
```

```
                    printf("- ");
                }

                        if(temp_inode->di_mode & DIFILE)
                {
                    printf("% ld",temp_inode->di_size);
                    printf(" block chain:");
                    for(k=0;k<temp_inode->di_size/BLOCKSIZ+1;k++)
                        printf("% d ",temp_inode->di_addr[k]);
                printf("\n");
                }
                else
                    printf("< dir> block chain:% d\n",dir.direct[i].d_ino);
                iput(temp_inode);
            }
        }
}

mkdir(dirname) /* mkdir */
char * dirname;
{
        int dirid,dirpos;
        struct inode * inode;
        struct direct buf[BLOCKSIZ/(DIRSIZ+2)];
        unsigned int block;

        dirid=namei(dirname);

        if(dirid != 0)
        {
            iode=iget(dirid);
            if(inode->di_mode & DIDIR)
                printf("\n% s directory already existed!\n",dirname);
            else
                printf("\n% s is a file name,&can't create a dir the same name\n",dirname);
            iput(inode);
            return;
        }

        dirpos=iname(dirname);
        inode=ialloc();
        dirid=inode->i_ino;
        dir.direct[dirpos].d_ino=inode->i_ino;
        dir.size++;
```

```
    /* fill the new dir buf */
    strcpy(buf[0].d_name,".");
    buf[0].d_ino=dirid;
    strcpy(buf[1].d_name,"..");
    buf[1].d_ino=cur_path_inode->i_ino;
    buf[2].d_ino=0;
    block=balloc();
    fseek(fd,DATASTART+block*BLOCKSIZ,SEEK_SET);
    fwrite(buf,1,BLOCKSIZ,fd);

    inode->di_size=2*(DIRSIZ+2);
    inode->di_number=1;
    inode->di_mode=user[user_id].u_default_mode|DIDIR;
    inode->di_uid=user[user_id].u_uid;
    inode->di_gid=user[user_id].u_gid;
    inode->di_addr[0]=block;
    iput(inode);
    return ;
}

chdir(dirname) /* chdir */
char *dirname;
{
    unsigned int dirid;
    struct inode *inod;
    unsigned short block;
    int i,j,low=0,high=0;
    dirid=namei(dirname);
    if(dirid==0)
    {
        printf("\n%s does not exist!\n",dirname);
        return;
    }

    inode=iget(dirid);
    if(!access(user_id,inode,user[user_id].u_default_mode))
    {
        printf("\nhas not access to the directory %s\n",dirname);
        iput(inode);
        return;
    }

    /* pack the current directory */
    for(i=0;i<dir.size;i++)
    {
```

```
        for (j= 0; j< DIRNUM; j+ + )
            if (dir.direct[j].d_ino= = 0)
                break;
            memcpy (&dir.direct[j], &dir.direct[i], DIRSIZ+ 2) ;
            dir.direct[j].d_ino= 0;
    }

        /* write back the current directory */
    for (i= 0; i< cur_path_inode-> di_size/BLOCKSIZ+ 1; i+ + )
    {
        bfree (cur_path_inode-> di_addr[i]) ;
    }

    for (i= 0; i< dir.size; i+ = BLOCKSIZ/(DIRSIZ+ 2))
    {
        block= balloc() ;
        cur_path_inode-> di_addr[i]= block;
        fseek (fd, DATASTART+ block * BLOCKSIZ, SEEK_SET) ;
        fwrite (&dir.direct[0], 1, BLOCKSIZ, fd) ;
    }

    cur_path_inode-> di_size= dir.size * (DIRSIZ+ 2) ;
    iput (cur_path_inode) ;

    cur_path_inode= inode;
        dir.size= inode-> di_size/(IRSIZ+ 2) ;

    /* read the change dir from disk */
    j= 0;
    for (i= 0; i< inode-> di_size/BLOCKSIZ+ 1; i+ + )
    {
        fseek (fd, DATASTART+ inode-> di_addr[i] * BLOCKSIZ, SEEK_SET) ;
        fread (&dir.direct[0], 1, BLOCKSIZ, fd) ;
        j+ = BLOCKSIZ/(DIRSIZ+ 2) ;
    }
    return;
}
```

(13) 文件创建函数 create()(文件名 create.c)

```
#include < stdio.h>
#include "filesys.h"

create (user_id, filename, mode)
unsigned int user_id;
char * filename;
```

```
unsigned short mode;
{
    unsigned int di_ino,di_ith;
    struct inode * inode;
    int i,j;

    di_ino=namei(filename);
    if(di_ino!=0)                          /* already exited */
    {
        inode=iget(di_ino);
        if(access(user_id,inode,mode)==0)
        {
            iput(inode);
            printf("\ncreate access not allowed");
            return;
        }

        /* free all the block of the old file */
        for(i=0;i<inode->di_size/BLOCKSIZ+1;i++)
        {
            bfree(inode->di_addr[i]);
        }

        /* to do: add code here to update the pointer of the sys_file */
        for(i=0;i<SYSOPENFILE;i++)
            if(sys_ofile[i].f_inode==inode)
                sys_ofile[i].f_off=0;

        for(i=0;i<NOFILE;i++)
            if(user[user_id].u_ofile[i]==SYSOPENFILE+1)
            {
                user[user_id].u_uid=inode->di_uid;
                user[user_id].u_gid=inode->di_gid;
                for(j=0;j<SYSOPENFILE;j++)
                    if(sys_ofile[j].f_count==0)
                    {
                        user[user_id].u_ofile[i]=j;
                        sys_ofile[j].f_flag=mode;
                    }
                    return i;
            }
    }
    else                                   /* not existed before */
    {
        inode=ialloc();
```

```
        di_ith=iname(filename);

        dir.size++;

        dir.direct[di_ith].d_ino=inode->i_ino;
        dir.direct[di_ith+1].d_ino=0;
        inode->di_mode=user[user_id].u_default_mode|DIFILE;
        inode->di_uid=user[user_id].u_uid;
        inode->di_gid=user[user_id].u_gid;
        inode->di_size=file_block;
        inode->di_number=1;

        for (i=0;i<SYSOPENFILE;i++)
            if(sys_ofile[i].f_count==0)
                break;

        for (j=0;j<NOFILE;j++)
            if(user[user_id].u_ofile[j]==SYSOPENFILE+1)
                break;

        user[user_id].u_ofile[j]=i;
        sys_ofile[i].f_flag=mode;
        sys_ofile[i].f_count=0;
        sys_ofile[i].f_off=0;
        sys_ofile[i].f_inode=inode;

        return j;
    }
}
```

(14) 打开文件函数 open()(文件名 open.c)

```
#include <stdio.h>
#include "filesys.h"

unsigned short open(user_id,filename,openmode)
int user_id;
char *filename;
unsigned short openmode;
{
    unsigned int dinodeid;
    struct inode *inode;
    int i,j;

    dinodeid=namei(filename);
    if(dinodeid==0)                          /* no such file */
```

```
{
      printf("\nfile does not exist!\n");
      return 0;
}
inode=iget(dinodeid);
if(!access(user_id,inode,openmode))                    /* access denied */
{
      printf("\nfile open has not access!\n");
      iput(inode);
      return 0;
}

/* alloc the sys_ofile item */
for(i=0;i<SYSOPENFILE;i++)
      if(sys_ofile[i].f_count==0) break;
if(i==SYSOPENFILE)
{
      printf("\nsystem open file too much!\n");
      iput(inode);
      return 0;
}
sys_ofile[i].f_inode=inode;
sys_ofile[i].f_flag=openmode;
sys_ofile[i].f_count=1;

if(openmode&FAPPEND)
      sys_ofile[i].f_off=inode->di_size;
else
      sys_ofile[i].f_off=0;

/* alloc the user open file item */
for(j=0;j<NOFILE;j++)
      if(user[user_id].u_ofile[j]==SYSOPENFILE+1)
            break;
if(j==NOFILE)
{
      printf("\nuser open file too much!\n");
      sys_ofile[i].f_count=0;
      iput(inode);
      return 0;
}

user[user_id].u_ofile[j]=1;
/* if APPEND, free the block of the file before */
if(!(openmode & FAPPEND)) {
```

```
        for(i=0;i<inode->di_size/BLOCKSIZ+1;i++)
            bfree(inode->di_addr[i]);
        inode->di_size=0;
    }
    return j;
}
```

(15) 关闭文件系统函数 close()(文件名 close.c)

```
#include <stdio.h>
#include "filesys.h"

close(user_id,cfd)               /* close */
unsigned int user_id;
unsigned short cfd;
{
    struct inode * inode;

    inode=sys_ofile[user[user_id].u_ofile[cfd]].f_inode;
    iput(inode);
    sys_ofile[user[user_id].u_ofile[cfd]].f_count--;
    user[user_id].u_ofile[cfd]=SYSOPENFILE+1;
}
```

(16) 删除文件函数 delete()(文件名 delete.c)

```
#include <stdio.h>
#include "filesys.h"

delete(filename)
char * filename;
{
    unsigned int dinodeid,i;
    struct inode * inode;

    dinodeid=namei(filename);
    if(dinodeid!=0)
        inode=iget(dinodeid);
    inode->di_number--;

    for(i=0;i<dir.size;i++)
        if(dir.direct[i].d_ino==dinodeid)
            break;
    i++;
    while(dir.direct[i].d_ino!=0)
    {
        strcpy(dir.direct[i-1].d_name,dir.direct[i].d_name);
```

```
            dir.direct[i-1].d_ino=dir.direct[i].d_ino;
            i++;
        }
        dir.direct[i-1].d_ino=0;
        dir.size=i-1;

        iput(inode);
        printf("\ndir.size=%d\n",dir.size);
}
```

(17) 读写文件函数 read()与 write()(文件名 rdwt.c)

```
#include <stdio.h>
#include "filesys.h"
#include "dirent.h"

unsigned int read(fd1,buf,size)
int fd1;
char *buf;
unsigned int size;
{
        unsigned long off;
        int block,block_off,i,j;
        struct inode *inode;
        char *temp_buf;

        inode=sys_ofile[user[user_id].u_ofile[fd1]].f_inode;
        if(!(sys_ofile[user[user_id].u_ofile[fd1]].f_flag&FREAD))
        {
            printf("\nthe file is not opened for read\n");
            return 0;
        }

        temp_buf=buf;

        off=sys_ofile[user[user_id].u_ofile[fd1]].f_off;
        if((off+size)>inode->di_size)
            size=inode->di_size-off;
        block_off=off%BLOCKSIZ;
        block=off/BLOCKSIZ;

        if(block_off+size<BLOCKSIZ)
        {
            fseek(fd,DATASTART+inode->di_addr[block]*BLOCKSIZ+block_off,SEEK_SET);
            fread(buf,1,size,fd);
            return size;
```

```
	}

	fseek(fd,DATASTART+inode->di_addr[block]*BLOCKSIZ+block_off,SEEK_SET);
	fread(temp_buf,1,BLOCKSIZ-block_off,fd);

	temp_buf+=BLOCKSIZ-block_off;
	j=(inode->di_size-off-block_off)/BLOCKSIZ;
	for(i=0;i<(size-block_off)/BLOCKSIZ;i++)
	{
		fseek(fd,DATASTART+inode->di_addr[j+i]*BLOCKSIZ,SEEK_SET);
		fread(temp_buf,1,BLOCKSIZ,fd);
		temp_buf+=BLOCKSIZ;
	}

	block_off=(size-block_off)%BLOCKSIZ;
	block=inode->di_addr[off+size/BLOCKSIZ+1];
	fseek(fd,DATASTART+block*BLOCKSIZ,SEEK_SET);
	fread(temp_buf,1,block_off,fd);

	sys_ofile[user[user_id].u_ofile[fd1]].f_off+=size;
	return size;
}

unsigned int write(fd1,buf,size)
int fd1;
char *buf;
unsigned int size;
{
	unsigned long off;
	int block,block_off,i,j,k=0;
	struct inode *inode;
	char *temp_buf;
		inode=sys_ofile[user[user_id].u_ofile[fd1]].f_inode;

	temp_buf=buf;

	off=sys_ofile[user[user_id].u_ofile[fd1]].f_off;
	block_off=off%BLOCKSIZ;
	block=off/BLOCKSIZ;

	if(block_off+size<BLOCKSIZ)
	{
		fseek(fd,DATASTART+inode->di_addr[block]*BLOCKSIZ+block_off,SEEK_SET);
		fwrite(buf,1,size,fd);
		printf("Write success!\n");
```

```
        return size;
    }

    if(sys_ofile[user[user_id].u_ofile[fd1]].f_flag==FAPPEND)
    {
        fseek(fd,DATASTART+inode->di_addr[block]*BLOCKSIZ+block_off,SEEK_SET);
        fwrite(temp_buf,1,BLOCKSIZ-block_off,fd);
        temp_buf+=BLOCKSIZ-block_off;
        k=1;
    }

    for(i=0;i<(size-k*(BLOCKSIZ-block_off))/BLOCKSIZ;i++)
    {
        inode->di_addr[block+1+i]=balloc();
        fseek(fd,DATASTART+inode->di_addr[block+k+i]*BLOCKSIZ,SEEK_SET);
        fwrite(temp_buf,1,BLOCKSIZ,fd);
        temp_buf+=BLOCKSIZ;
    }

    block_off=(size-k*(BLOCKSIZ-block_off))%BLOCKSIZ;
    block=inode->di_addr[block+k+i]=balloc();
    fseek(fd,DATASTART+block*BLOCKSIZ,SEEK_SET);
    fwrite(temp_buf,1,block_off,fd);
    sys_ofile[user[user_id].u_ofile[fd1]].f_off+=size;
    inode->di_size=sys_ofile[user[user_id].u_ofile[fd1]].f_off;
    return size;
}
```

(18) 注册和退出函数 login()和 logout()(文件名 log.c)

```
#include <stdio.h>
#include "filesys.h"

int login(uid,passwd)                                   /* login() */
unsigned short uid;
char *passwd;
{
    int i,j;

    for(i=0;i<PWDNUM;i++)
    {
        if((uid==pwd[i].p_uid)&&!(strcmp(passwd,pwd[i].password)))
        {
            for(j=0;j<USERNUM;j++)
                if(user[j].u_uid==0)
                    break;
```

```
            if(j==USERNUM)
            {
                printf("\ntoo much user in the system,waited to login\n");
                return 0;
            }
            else
            {
                user[j].u_uid=uid;
                user[j].u_gid=pwd[i].p_gid;
                user[j].u_default_mode=DEFAULTMODE;
            }
            break;
        }
    }
    if(i==PWDNUM)
    {
        printf("\nincorrect password,Login Failure!\n");
        return 0;
    }
    else
    {
        printf("Login Success!%d's user id is %d\n",uid,j);
        return 1;
    }
}

int logout(uid)                              /* logout() */
unsigned short uid;
{
    int i,j,sys_no;
    struct inode *inode;

    for(i=0;i<USERNUM;i++)
        if(uid==user[i].u_uid)
            break;
    if(i==USERNUM)
    {
        printf("\nno such a file\n");
        return 0;
    }

    for(j=0;j<NOFILE;j++)
    {
        if(user[i].u_ofile[j]!=SYSOPENFILE+1)
        {
```

```
                sys_no=user[i].u_ofile[j];
                inode=sys_ofile[sys_no].f_inode;
                iput(inode);
                sys_ofile[sys_no].f_count--;
                user[i].u_ofile[j]=SYSOPENFILE+1;
            }
        }
        printf("no user in the file system \n");
        return 1;
}
```

(19) 打印输出函数 dirlt()(文件名 dirlt.c)

```
#include <stdio.h>
#include "filesys.h"

dirlt(j)
int j;
{   int i;

    printf("dir.size=%d\n",dir.size);
    for(i=0;i<dir.size+j;i++)
    {printf("i=%d,d_name=%s,d_ino=%d\n",i,dir.direct[i].d_name,dir.direct[i].
    d_ino);
    }
}
```

【结果】

对上述 makefile 文件进行编译后可得执行文件 filsys。在 Linux 或 UNIX SystemⅤ以上版本环境下,运行 filsys,可对上述文件系统程序进行测试。其结果如下:

```
$ filsys <CR>

$ format
$ Do you want to format the disk?
  y
$ Format will erase all context on the disk. Are You Sure?
  y

$ install

$ dir
    CURRENT DIRECTORY:
    .              d xxx xxx xxx <dir> block chain:1
    ..             d xxx xxx xxx <dir> block chain:2
```

```
$ login(2118,"abcd")

$ mkdir a2118
$ chdir a2118

$ create(user_id, "file0.c", 01777)
$ write (ab_fd1, buf, 3077)
$ close (user_id,ab_fd1)

$ mkdir subdir
$ chdir subdir

$ create(user_id, "file1.c", 01777)
$ write (ab_fd2, buf, 2068)

$ chdir ..

$ create(user_id, "file2.c", 01777)
$ write(ab_fd3,buf,1791)
$ close (user_id,ab_fd3)

$ dir
    CURRENT DIRECTORY;
    .              d xxx xxx xxx <dir> block chain:1
    ..             d xxx xxx xxx <dir> block chain:2
    file0.c        f xxx ____ ____ 3077 block chain:4 5 6 7 8 9 10
    subdir         d xxx xxx xxx <dir> blokc chain:11
    file2.c        f xxx ____ ____ 1791 block chain: 17 18 19 20

$ delete file0.c

$ create(user_id, "file3.c", 01700)
$ write(ab_fd4, buf, 4396)
$ close(user_id,ab_fd4)

$ dir
    CURRENT DIRECTORY:
    .              d xxx xxx xxx <dir> block chain:1
    ..             d xxx xxx xxx <dir> block chain:2
    subdir         d xxx xxx xxx <dir> block chain:11
    file2.c        f xxx ____ ____ 1791 block chain: 17 18 19 20
    file3.c        f xxx ____ ____ 4396 block chain:4 5 6 7 8 9 10 21 22 23

$ open(user_id, "file2.c", 03)
$ write(ab_fd3, buf, 1636)
```

```
$ close(user_id,ab_fd3)

$ dir
    CURRENT DIRECTORY:
    .               d xxx xxx xxx <dir> block chain:1
    ..              d xxx xxx xxx <dir> block chain:2
    subdir          d xxx xxx xxx <dir> block chain:11
    file2.c         f xxx ___ ___ 3427 block chain: 17 18 19 20 24 25 26
    file3.c         f xxx ___ ___ 4396 block chain: 4 5 6 7 8 9 10 21 22 23

$ chdir ..

$ logout (2118)
  no user in the file system.

$ halt
  Goodbye. See you next time. Please turn off the switch.
```

注意,本文件系统程序未使用命令交互解释工具 Shell,读者可从本程序中观察到这一点。